Spine Toolbox Documentation

Release 0.6.13.dev0-dev.0

Spine project consortium

Nov 02, 2022

CONTENTS:

Spine Toolbox is an application, which provides means to define, manage, and execute complex data processing and computation tasks, such as energy system models.

If you are new to Spine Toolbox, *Getting Started* section is a good place to start. If you want to run SpineOpt.jl using Spine Toolbox, *How to set up SpineOpt.jl* provides the step-by-step instructions on how to get started. For information on how to set up Python, Julia, and Gams for Spine Toolbox, see *Setting up External Tools*. Please see *Settings* chapter for information on user customizable Spine Toolbox settings. If you need help in understanding the terms we use throughout the app and this User Guide, please check the *Terminology* section. If you want to contribute to this project, please see the *Contribution Guide for Spine Toolbox*. The last section contains the complete code reference of Spine Toolbox.

CHAPTER

ONE

GETTING STARTED

Welcome to the Spine Toolbox's getting started guide. In this guide you will learn two ways of running a "Hello, World!" program on Spine Toolbox. If you need help on how to run **SpineOpt.jl** using Spine Toolbox, see chapter *How to set up SpineOpt.jl*.

This chapter introduces the following topics:

1.1 Spine Toolbox Interface

The central element in Spine Toolbox's interface is the *Design View*, which allows you to visualize and manipulate your project workflow. In addition to the *Design View* there are a few 'dock widgets' that provide additional functionality:

- *Project* provides a more concise view of your project, including the *Items* that are currently in the project, grouped by category: Data Stores, Data Connections, Tools, Views, Importers, Exporters and Manipulators.
- *Properties* provides an interface to interact with the currently selected project item.
- *Event Log* shows relevant messages about user performed actions and the status of executions.
- *Item Execution Log* shows the output of executed project items.
- *Python console* provides an interface to interact with the Python programming language, and also allows Spine Toolbox to execute Python Tools.
- *Julia console* provides an interface to interact with the Julia programming language, and also allows Spine Toolbox to execute Julia Tools.
- *Executions* shows a list of parallel executions available in the project.

In addition to the Design view and the dock widgets, the main window contains a *toolbar* split into two sections. The *Main* section contains the project items that you can drag-and-drop onto the Design View and the *Execute* section has buttons related to executing the project.

Tip: You can drag-and-drop the dock widgets around the screen, customizing the interface at your will. Also, you can select which ones are shown/hidden using either the **View/Dock Widgets** menu, or the main menu toolbar's context menu. Spine Toolbox remembers your configuration between sessions. Selecting **Restore Dock Widgets** from the **View/Dock Widgets** menu restores the widgets back to their default location.

Tip: Most elements in the Spine Toolbox's interface are equipped with *tool tips*. Leave your mouse cursor over an element (button, checkbox, list, etc.) for a moment to make the tool tip appear.

1.2 Creating a Project

To create a new project, please do one of the following:

- From the application main menu, select **File -> New project…**
- Press *Ctrl+N*.

The *Select project directory (New project…)* dialog will show up. Browse to a folder of your choice and create a new directory called 'hello world' there. Then select the 'hello world' directory and press Enter. Spine Toolbox will populate the selected directory with some files and directories it needs to store the project's data.

Congratulations, you have created your first Spine Toolbox project.

1.3 Creating a Tool specification

Note: Spine Toolbox is designed to run and connect multiple tools, which are specified using **Tool specifications**. You may think of a Tool specification as a self-contained program specification including a list of source files, required and optional input files, and expected output files. Once a Tool specification is added to a project, it can then be associated to a **Tool** item for its execution as part of the project workflow.

Note: Just like the main window, the Tool specification editor consists of dock widgets that you can reorganize however you like.

In the *toolbar*, click on the icon next to the Tool icon , to reveal the Tool specification list. Since there are none in the project yet, click on the button to open the *Tool specification editor*. Follow the instructions below to create a minimal Tool specification:

- Type 'hello_world' into the *Name:* field.
- Select 'Python' from the *Tool type* dropdown list,
- Click on the button next to the *Main program file* text in the *Program files* dock widget. A *Create new main program file* file browser dialog opens. Name the file *hello_world.py* and save it e.g. directly to the 'hello world' project directory or to a folder of your choice.

We have just created a 'hello_world.py' Python script file, but at the moment the file is empty. Spine Toolbox provides an mini **IDE** where you can view and edit the contents of Tool specification files. Let's try it out.

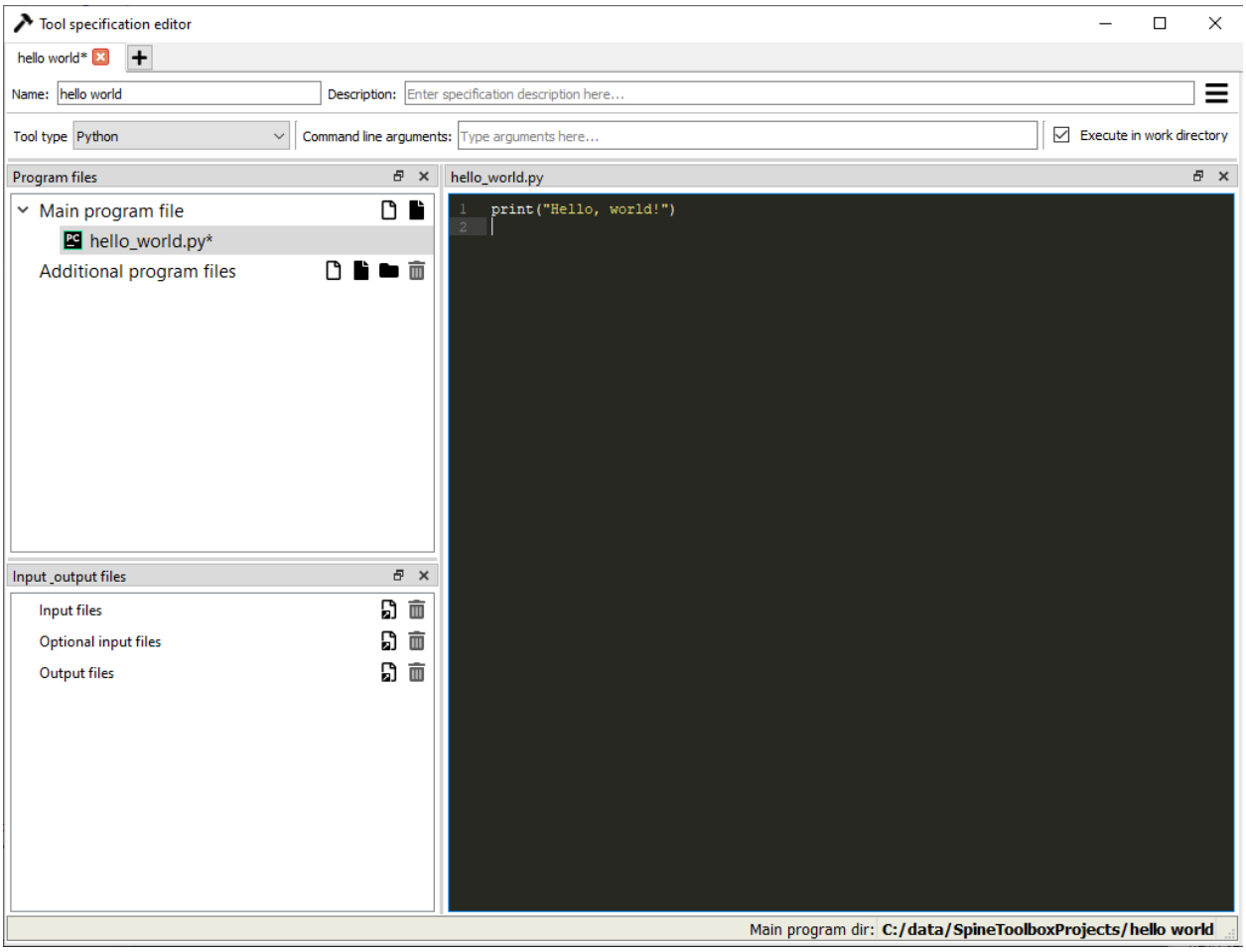
Select 'hello_world.py' below the *Main Program File*. Click on the (black) editor dock widget with the title 'hello_world.py'.

Type in the following:

```
print("Hello, world!")
```

Now, whenever *hello_world.py* is executed, the sentence 'Hello, World!' will be printed to the standard output.

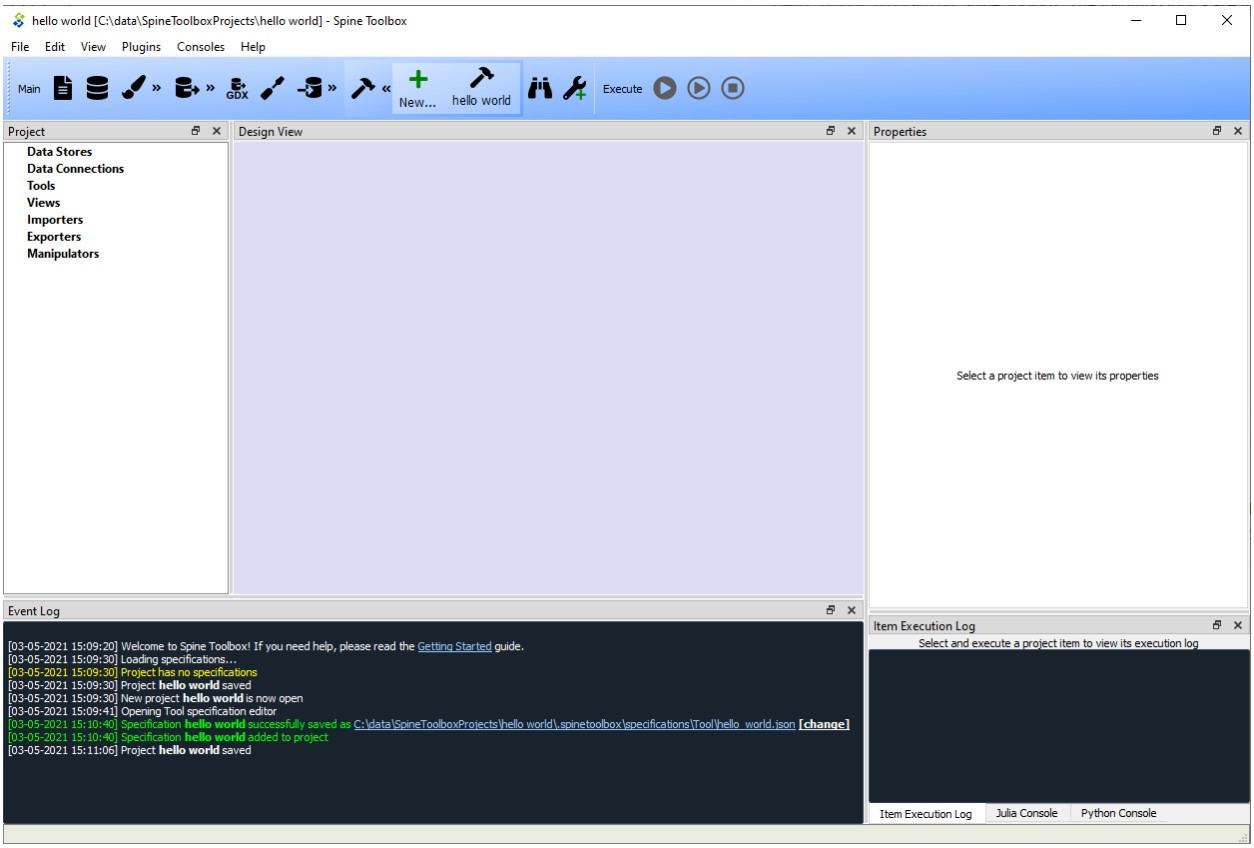
The *Tool specification editor* should be looking similar to this now:



Note that the program file (hello_world.py) and the Tool specification (hello world) now have unsaved changes. This is indicated by the star (*) character next to hello_world.py* and the Tool specification name in the tabbar (hello world*).

- Save changes to both by either pressing **Ctrl-s** or by mouse clicking on **Save** in the hamburger menu in the upper right hand corner.
- Close Tool specification editor by pressing **Alt-F4** or by clicking on 'X' in the top right hand corner of the window.

Your main window should look similar to this now.



Tool specifications are saved in JSON format by default into a dedicated directory under the project directory. If you want you can open the newly created hello_world.json file by clicking on the file path in the Event log message. The file will open in an external editor provided that you have selected a default program for files with the .json extension (e.g in Windows 10 you can do this in Windows Settings->Apps->Default apps). In general, you don't need to worry about *the contents* of the JSON Tool specification files. Editing these is done under the hood by the app.

If you want to save the 'hello_world.json' file somewhere else, you can do this by clicking the white [Change] link after the path in the Event Log.

Tip: Saving the Tool specification into a file allows you to add and use the same Tool specification in another project. To do this, you just need to click *add tool specification from file...* button () in the toolbar and select the tool specification file (.json) from your system.

Congratulations, you have just created your first Tool specification.

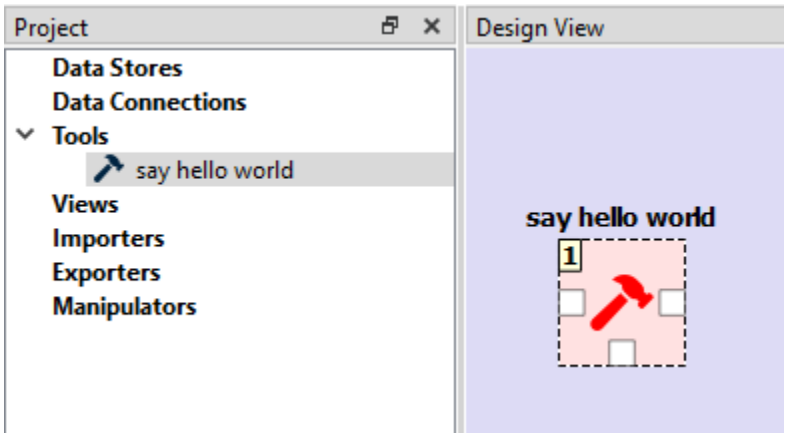
1.4 Adding a Tool item to the project

Note: The **Tool** project item is used to run Tool specifications.

Let's add a Tool item to our project, so that we're able to run the Tool specification we created above. To add a Tool item drag-and-drop the Tool icon from the toolbar onto the *Design View*.

The *Add Tool* form will popup. Change name of the Tool to 'say hello world', and select 'hello_world' from the dropdown list just below, and click **Ok**. Now you should see the newly added Tool item as an icon in the *Design View*,

and also as an entry in the *Project* dock widget, *Items* list, under the 'Tools' category. It should look similar to this:

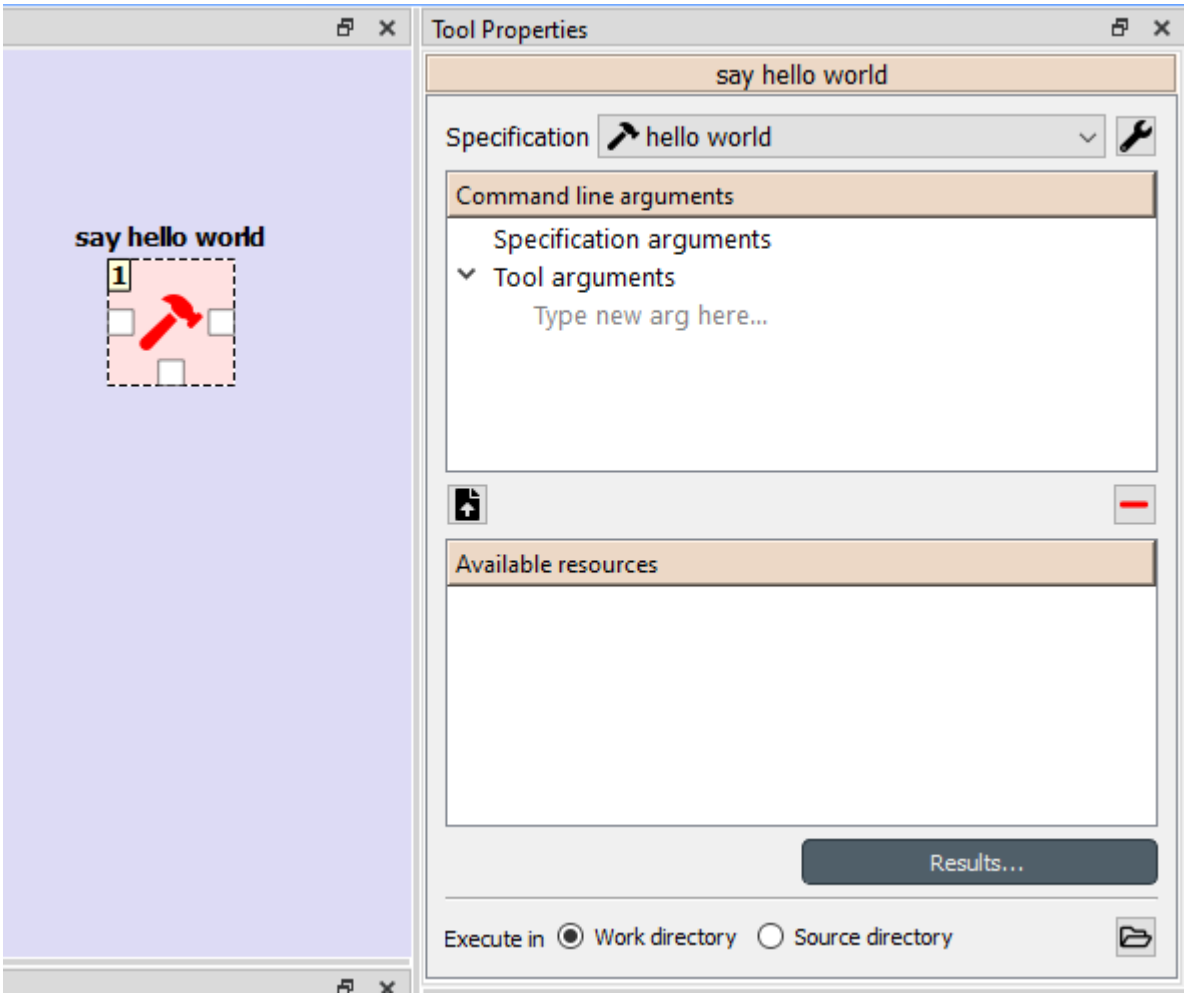


Another way to do the same thing is to drag the with the 'hello world' text from the toolbar onto the Design View. Similarly, the *Add Tool* form will popup but the 'hello world' tool specification is already selected from the dropdown list.

Note: The Tool specification is now saved to disk but the project itself is not. Remember to save the project every once in a while when you are working. You can do this from the main window *File->Save project* button or by pressing **Ctrl-s** when the main window is active.

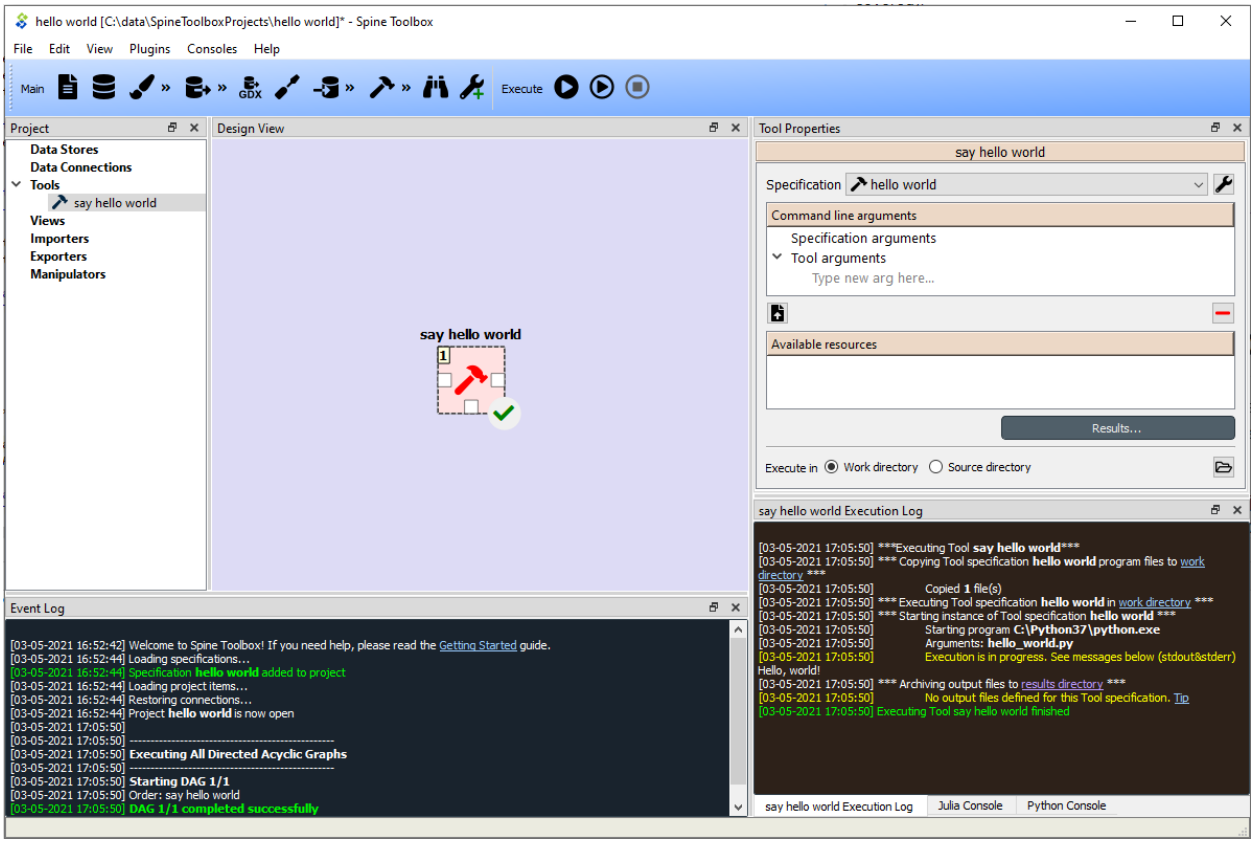
1.5 Executing a Tool

Select the 'say hello world' Tool on *Design View*, and you will see its *Properties* in the dedicated dock widget. It looks similar to this:



Press *execute project* button on the toolbar. This will execute the 'say hello world' Tool project item which now has the 'hello world' Tool specification associated to it. In actuality, this will run the main program file *hello_world.py* in a dedicated process.

Once the execution is finished, you can see the item execution details in the *Item Execution Log* and the details about the whole execution in Event Log.



Note: For more information about execution modes in Spine Toolbox, please see *Setting up External Tools* for help.

Congratulations, you just executed your first Spine Toolbox project.

1.6 Editing a Tool specification

To make things more interesting, we will now specify an *input file* for our 'hello_world' Tool specification.

Note: Input files specified in the Tool specification can be used by the program source files, to obtain input data for the Tool's execution. When executed, a Tool item looks for input files in **Data Connection**, **Data Store**, **Gdx Exporter**, **Exporter**, and **Data Transformer** project items connected to its input.

Open the Tool specification editor for the 'hello world' Tool spec. You can do this for example, by double-clicking the 'say hello world' Tool, or by selecting **Edit specification** from the 'hello world' Tool specification context menu in the toolbar, or from the 'say hello world' Tool context-menu (**Specification...->Edit specification**).

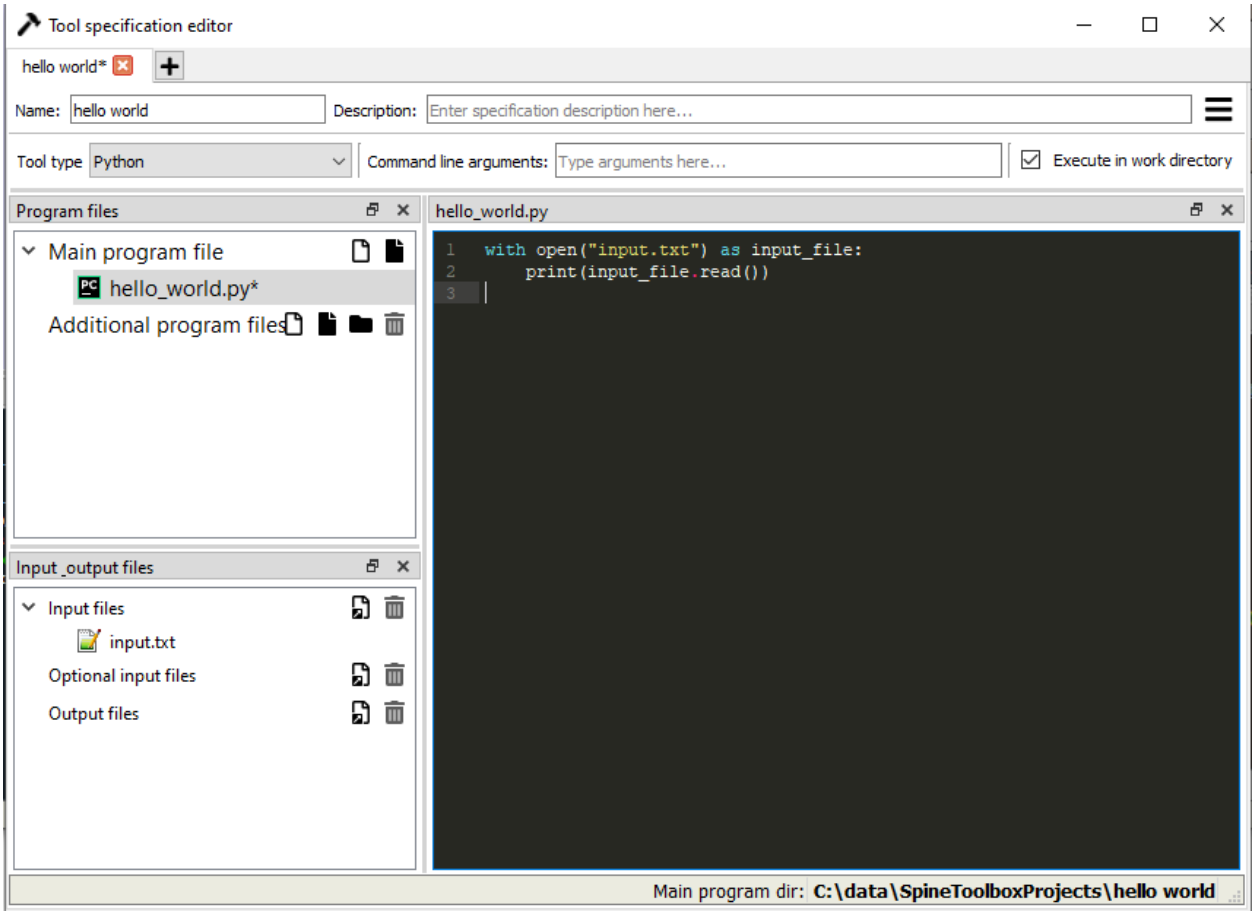
In *Input & Output files* dock widget, click the button next to the *Input Files* text. A dialog appears, that lets you enter a name for an input file. Type 'input.txt' and press Enter.

So far so good. Now let's use this input file in our program. Still in the Tool specification editor, replace the text in the main program file (hello_world.py), with the following:

```
with open("input.txt") as input_file:
    print(input_file.read())
```

Now, whenever *hello_world.py* is executed, it will look for a file called 'input.txt' in the current directory, and print its content to the standard output.

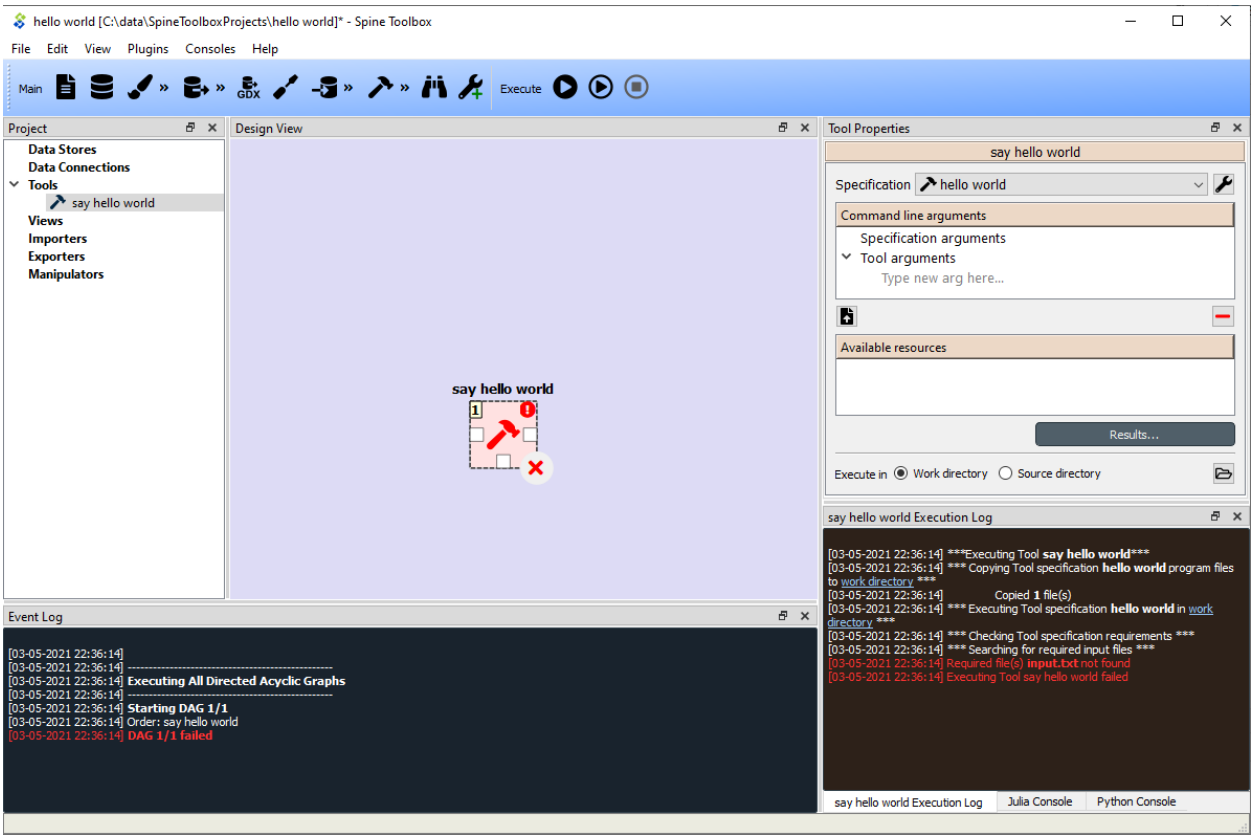
The editor should now look like this:



Save the specification and close the editor by pressing **Ctrl-s** and then **Alt-F4**.

Note: See *Tool specification editor* for more information on editing Tool specifications.

Back in the main window, note the exclamation mark on the Tool icon in Design View, if you hover the mouse over this mark, you will see a tooltip telling you in detail what is wrong. If you want you can try and execute the Tool anyway by pressing in the toolbar. *The execution will fail.* because the file 'input.txt' is not made available for the Tool:



1.7 Adding a Data Connection item to the project

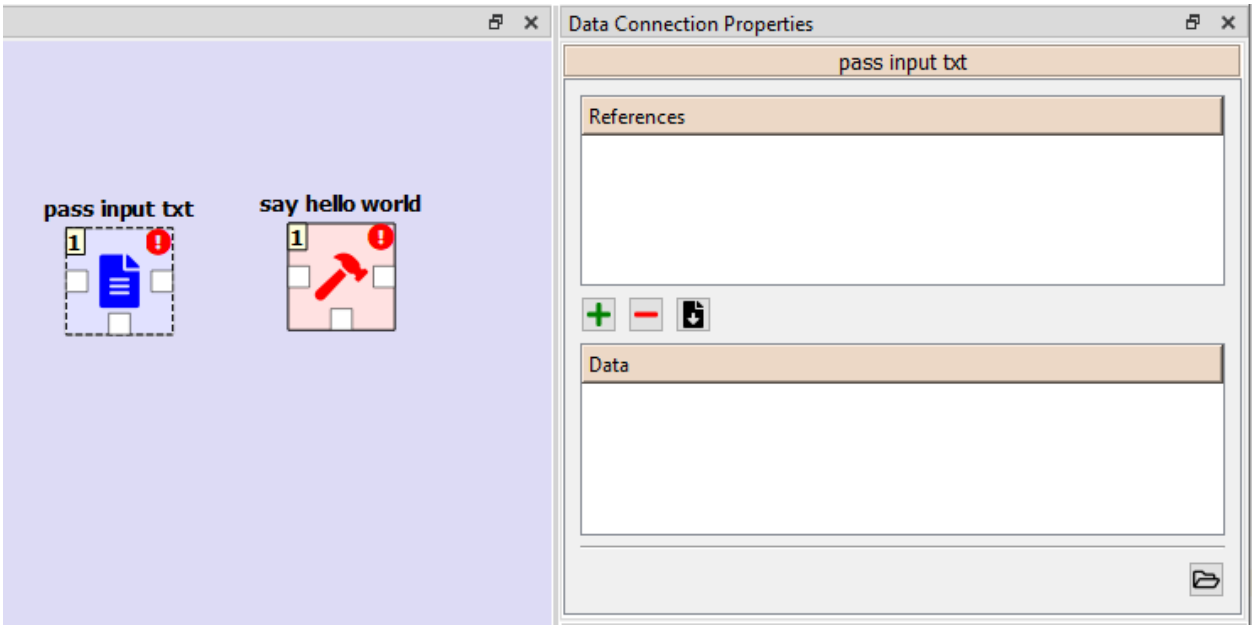
Note: The **Data Connection** item is used to hold generic data files, so that other items, notably Importer and Tool items, can make use of that data.

Let's add a **Data Connection** item to our project, so that we're able to pass the file 'input.txt' to 'say hello world'. To add a Data Connection item, drag-and-drop the Data Connection icon () from the toolbar onto the *Design View*.

The *Add Data Connection* form will show up. Type 'pass input txt' in the name field and click **Ok**. The newly added Data Connection item is now in the *Design View*, and also as an entry in the *Project* dock widgets items list, under the 'Data Connections' category. It should look similar to this:

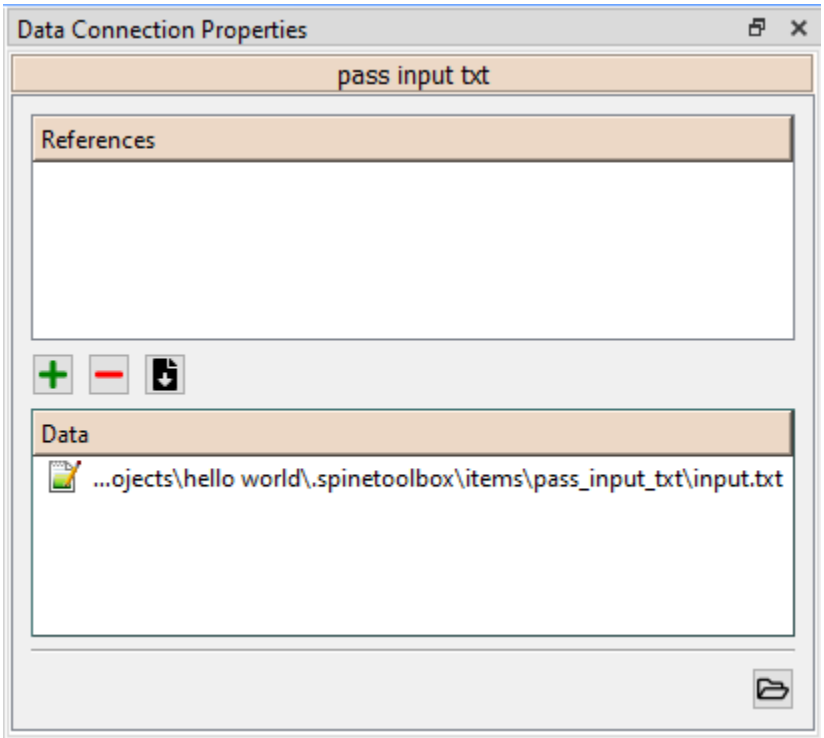
1.8 Adding data files to a Data Connection

Select the 'pass input txt' Data Connection item to view its properties in the *Properties* dock widget.



Right click anywhere within the *Data* box and select **New file…** from the context menu. When prompted to enter a name for the new file, type 'input.txt' and click **Ok**.

There's now a new file in the *Data* list:



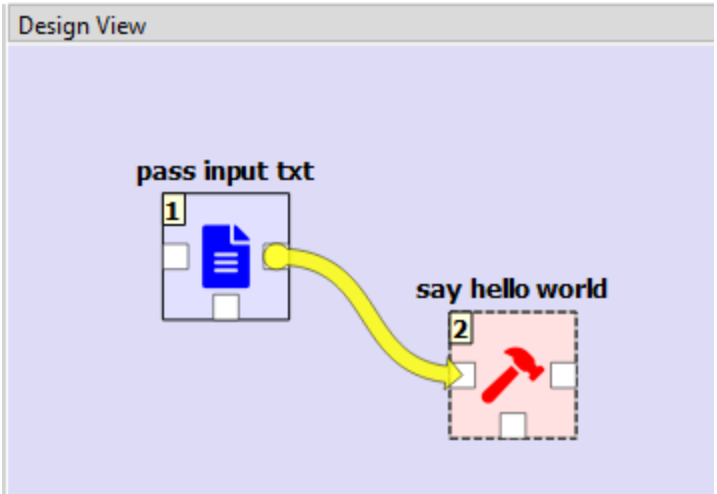
Double click this file to open it in your default text editor. Then enter the following into the file's content:

```
Hello again, World!
```

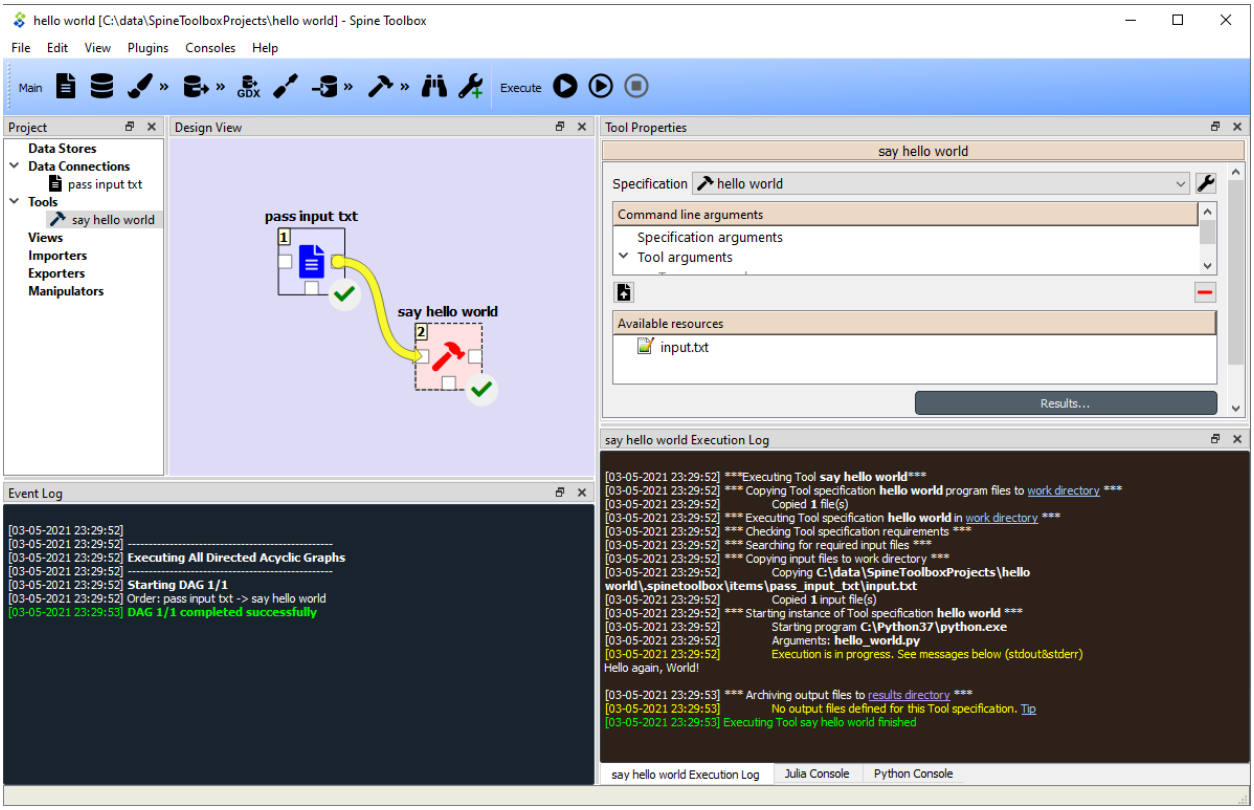
Save the file.

1.9 Connecting project items

As mentioned above, a Tool item looks for input files in Data Connections connected to its input. Thus you now need to create a connection from 'pass input txt' to 'say hello world'. To do this, click on one of the *connector* slots at the edges of 'pass input txt' in the *Design view*, and then on a similar slot in 'say hello world'. This will create an arrow pointing from one to another, as seen below:



Press once again. The project will be executed successfully this time:



That's all for now. I hope you've enjoyed following this guide as much as I enjoyed writing it. See you next time.

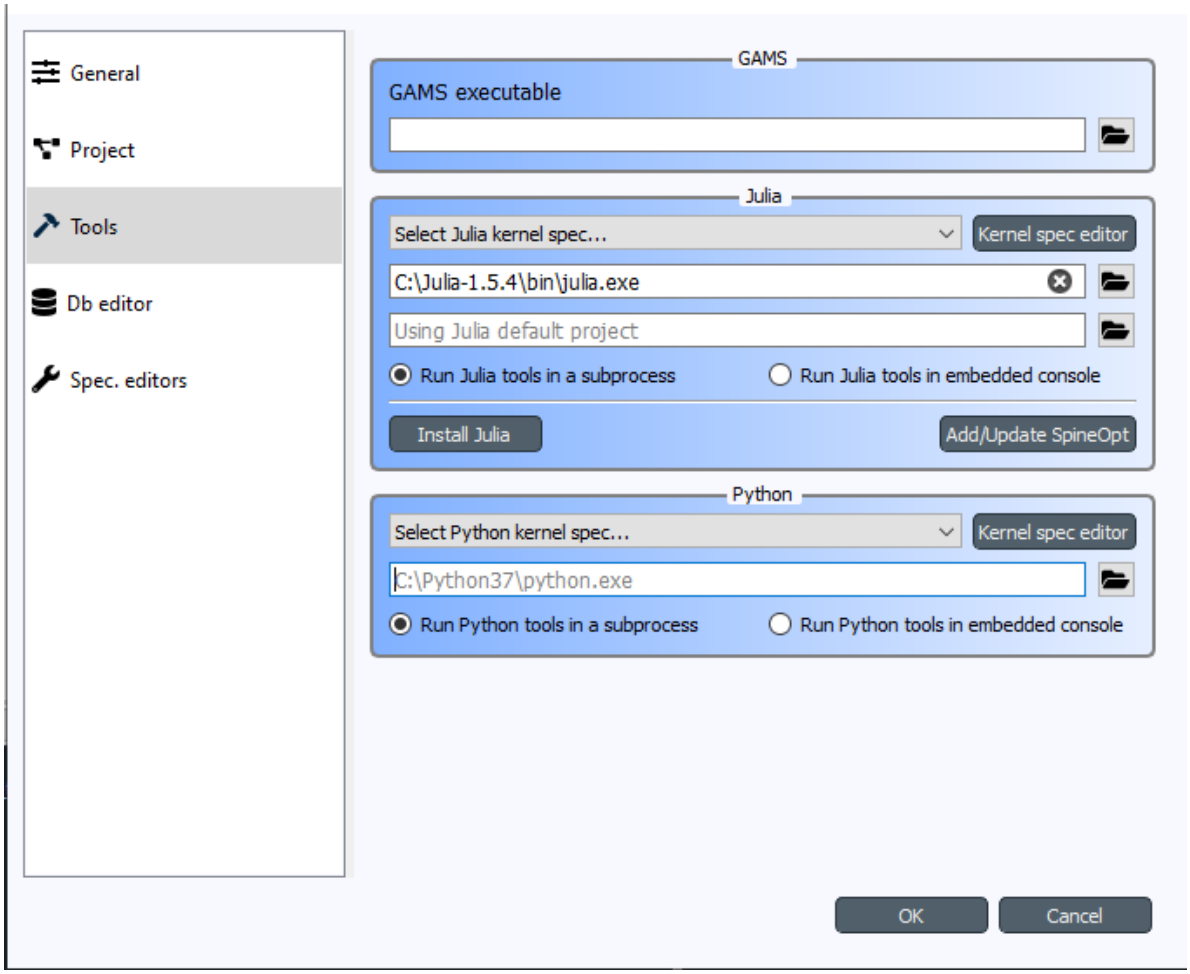
Where to next: If you need help on how to set up and run **SpineOpt.jl** using Spine Toolbox, see chapter *How to set up SpineOpt.jl*.

CHAPTER

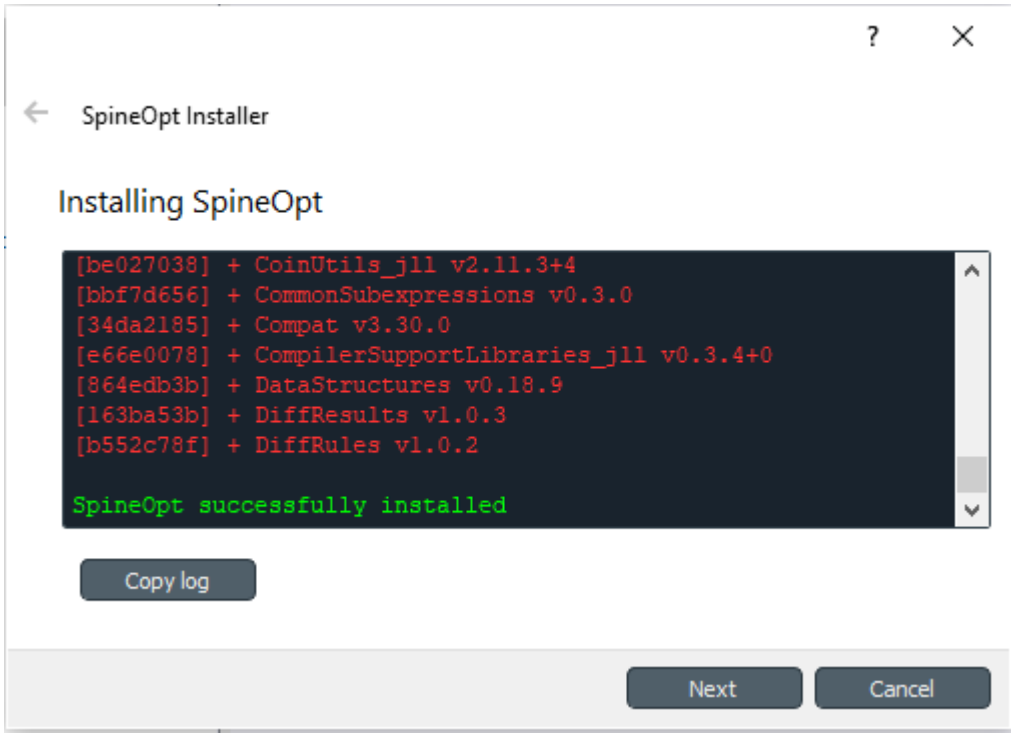
TWO

HOW TO SET UP SPINEOPT.JL

1. Install Julia (v1.2 or later) from https://julialang.org/downloads/ if you don't have one. See latest **SpineOpt.jl** Julia compatibility information here.
2. Start Spine Toolbox
3. Create a new project (*File->New project...*)
4. Select *File->Settings* from the main menu and open the *Tools* page.
5. Set a path to a Julia executable to the appropriate line edit (e.g. *C:\Julia-1.5.4\bin\julia.exe*). Your selections should look similar to this now.



6. *[Optional]* If you want to install and run SpineOpt in a specific Julia project environment (the place for Project.toml and Manifest.toml), you can set the path to the environment folder to the line edit just below the Julia executable (the one that says *Using Julia default project*).
7. Next, you need to install **SpineOpt.jl** package for the Julia you just selected for Spine Toolbox. You can do this manually by following the instructions **or** you can install **SpineOpt.jl** by clicking the *Add/Update SpineOpt* button. After clicking the button, an install/upgrade Spineopt wizard appears. Click *Next* twice and finally *Install SpineOpt*. **Wait until the process has finished** and you are greeted with this screen.



   Close the wizard.
8. Click Ok to close the *Settings* window
9. Back in the main window, select *PlugIns->Install plugin...* from the menu
10. Select *SpineOpt* and click Ok. After a short while, a red *SpineOpt Plugin Toolbar* appears on the main window.

Spine Toolbox and Julia are now correctly set up for running **SpineOpt.jl**. Next step is to Create a project workflow using SpineOpt.jl (takes you to SpineOpt documentation). See also *Tutorials* for more advanced use cases. For more information on how to select a specific Python or Julia version, see *Setting up External Tools*).

Note: The *SpineOpt Plugin Toolbar* contains two predefined Tools that make use of SpineOpt.jl. **The SpineOpt Plugin is not a requirement to run SpineOpt.jl**, they are provided just for convenience and as examples to get you started quickly.

CHAPTER

THREE

TUTORIALS

Welcome to the Spine Toolbox's tutorials page. The following tutorials are available:

3.1 Simple System tutorial

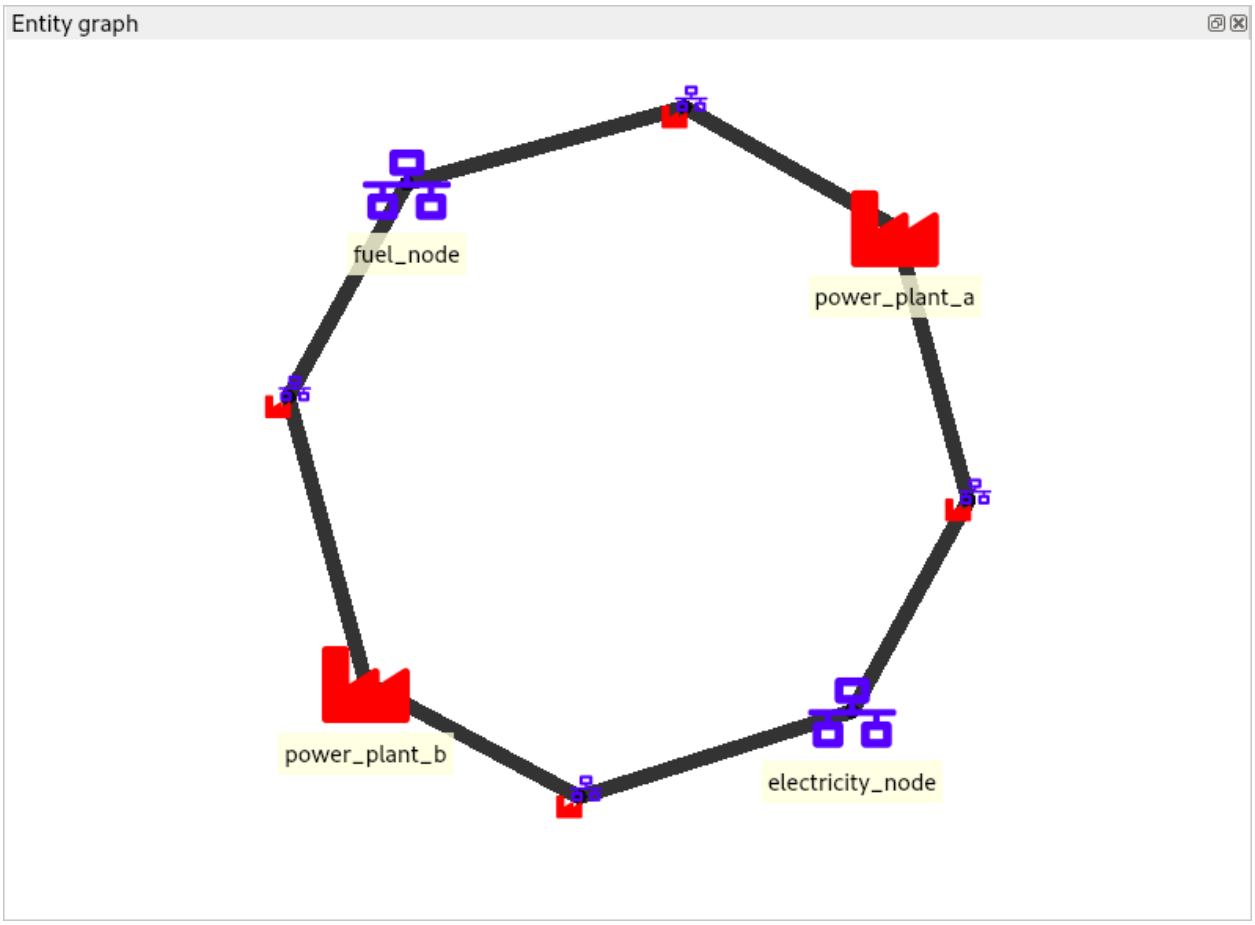
Welcome to Spine Toolbox's Simple System tutorial.

This tutorial provides a step-by-step guide to setup a simple energy system on Spine Toolbox and is organized as follows:

3.1.1 Introduction

Model assumptions

- Two power plants take fuel from a source node and release electricity to another node in order to supply a demand.
- Power plant 'a' has a capacity of 100 MWh, a variable operating cost of 25 euro/fuel unit, and generates 0.7 MWh of electricity per unit of fuel.
- Power plant 'b' has a capacity of 200 MWh, a variable operating cost of 50 euro/fuel unit, and generates 0.8 MWh of electricity per unit of fuel.
- The demand at the electricity node is 150 MWh.
- The fuel node is able to provide infinite energy.



3.1.2 Guide

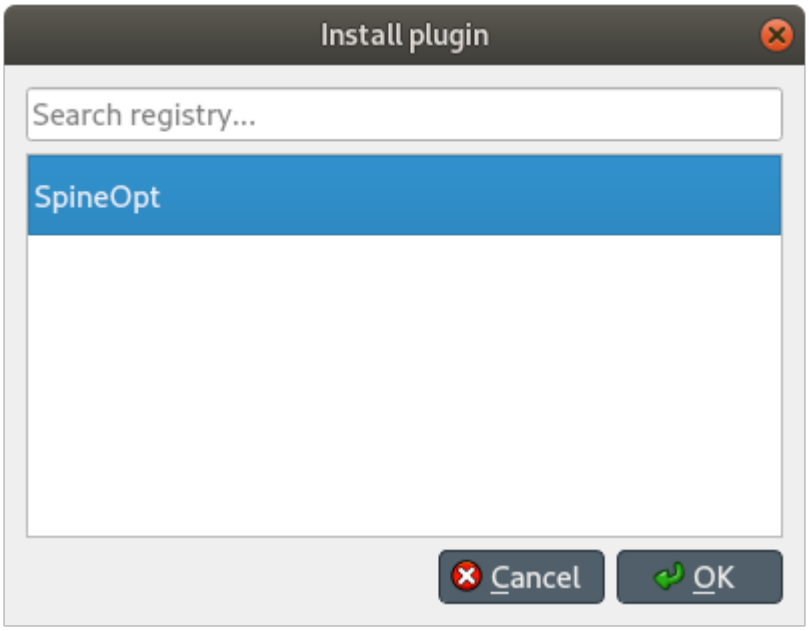
Installing requirements

Note: This tutorial is written for latest Spine Toolbox and SpineOpt development versions.

- If you haven't, follow the instructions here to install Spine Toolbox and SpineOpt in your system.
- If you already have Spine Toolbox and SpineOpt installed, please follow the instructions here and here to upgrade to the latest versions.

Installing the SpineOpt plugin

1. Launch Spine Toolbox and select **Plugins -> Install plugin…** from the main menu. The *Install plugin* dialog will pop up. Select SpineOpt from the list and press **Ok**.



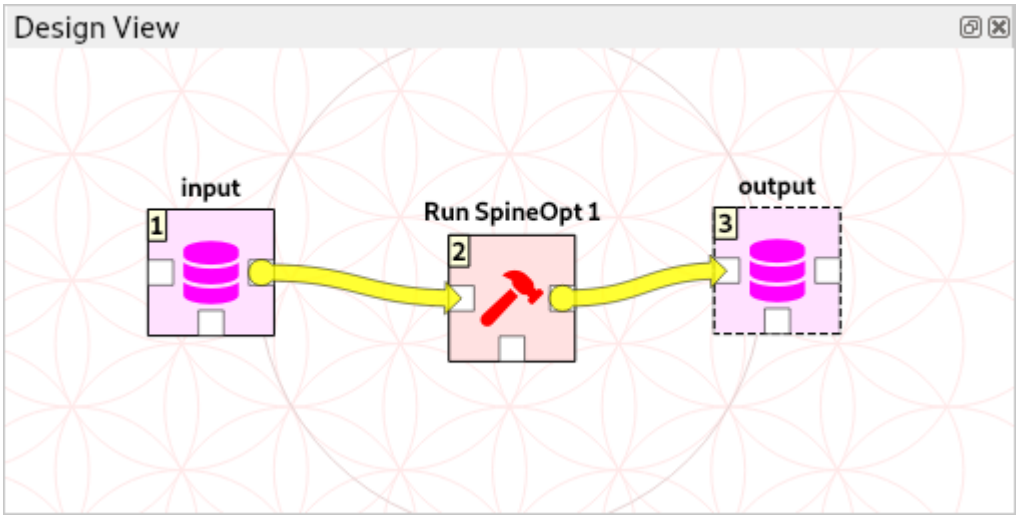
A new toolbar will appear, looking similar to this:

Setting up project

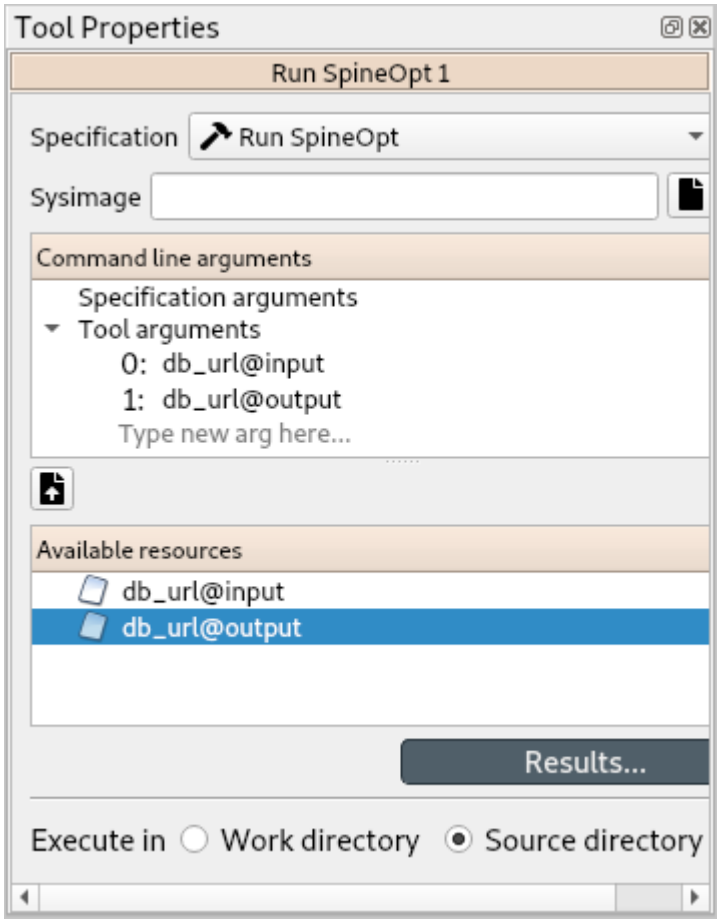
1. Select **File -> New project…** from Spine Toolbox main menu. Browse to a location where you want to create the project and create a new folder for it, called e.g. 'SimpleSystem', and then click **Open**.
2. Drag the *Data Store* icon from the tool bar and drop it into the *Design View*. This will open the *Add Data Store* dialog. Type 'input' as the Data Store name and click **Ok**.
3. Repeat the above procedure to create a Data Store called 'output'.
4. Create a database for the 'input' Data Store:
   1. Select the *input* Data Store item in the *Design View* to show the *Data Store Properties* (on the right side of the window, usually).
   2. In *Data Store Properties*, select the *sqlite* dialect at the top, and hit **New Spine db**. A dialog will pop up to let you select a name for the database file; just accept the default name.
5. Repeat the above procedure to create a database for the 'output' Data Store.
6. Drag the *Run SpineOpt* icon from the SpineOpt tool bar into the *Design View*. This will open the *Add Tool* dialog. Accept the default name ('Run SpineOpt 1') and click **Ok**.

   Note: Each item in the *Design view* is equipped with three *connectors* (the small squares at the item boundaries).

7. Click on one of 'input' connectors to start a *connection*, and then on one of 'Run SpineOpt 1' connectors to close it.
8. Repeat the procedure to create a *connection* from 'Run SpineOpt 1' to 'output'. It should look something like this:



9. Setup the arguments for the *Run SpineOpt* Tool:
   1. Select the *Run SpineOpt* Tool to show the *Tool Properties* (on the right side of the window, usually). You should see two elements listed under *Available resources*, `{db_url@input}` and `{db_url@output}`.
   2. Drag the first resource, `{db_url@input}`, and drop it in *Command line arguments*; then drag the second resource, `{db_url@output}`, and drop it right below the previous one. The panel should be now looking like this:



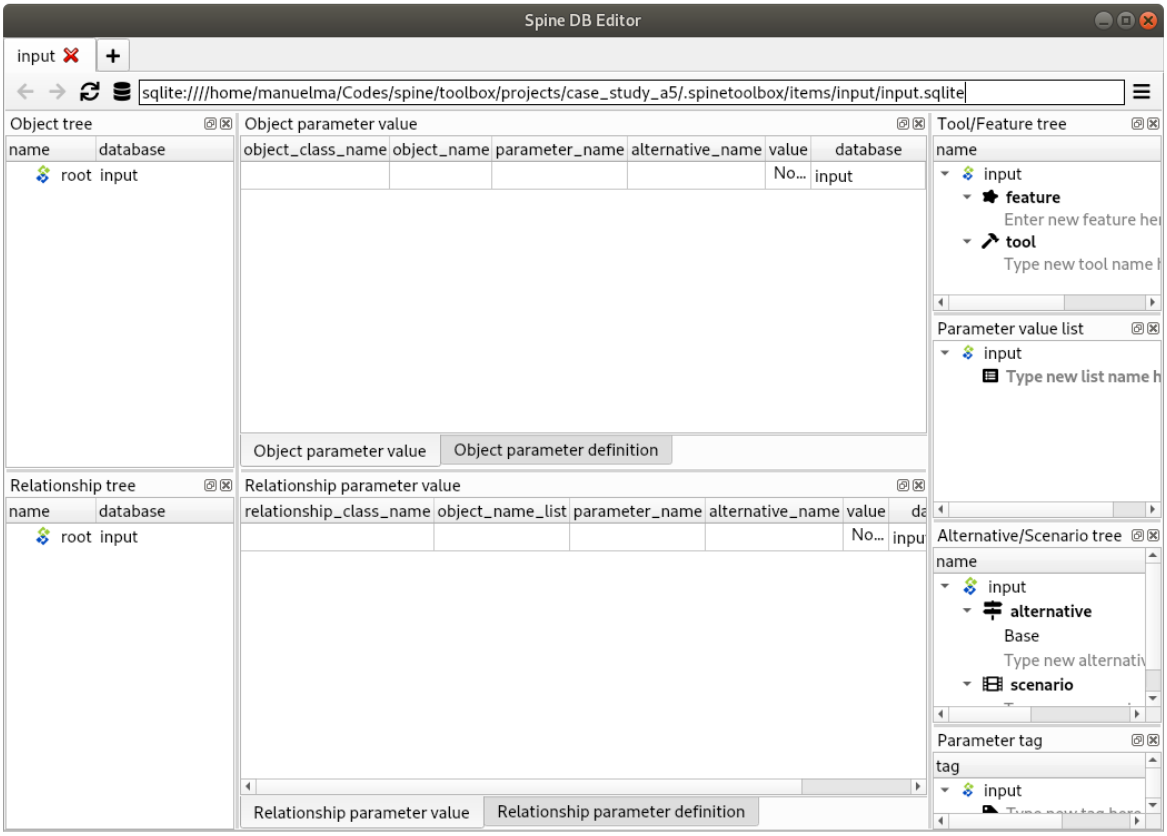
3. Double-check that the *order* of the arguments is correct: first, `{db_url@input}`, and second, `{db_url@output}`. (You can drag and drop to reorganize them if needed.)

10. From the main menu, select **File -> Save project**.

Entering input data

Importing the SpineOpt database template

1. Download the SpineOpt database template and the basic SpineOpt model (right click on the links, then select *Save link as…*)
2. Select the 'input' Data Store item in the *Design View*.
3. Go to *Data Store Properties* and hit **Open editor**. This will open the newly created database in the *Spine DB editor*, looking similar to this:



Note: The *Spine DB editor* is a dedicated interface within Spine Toolbox for visualizing and managing Spine databases.

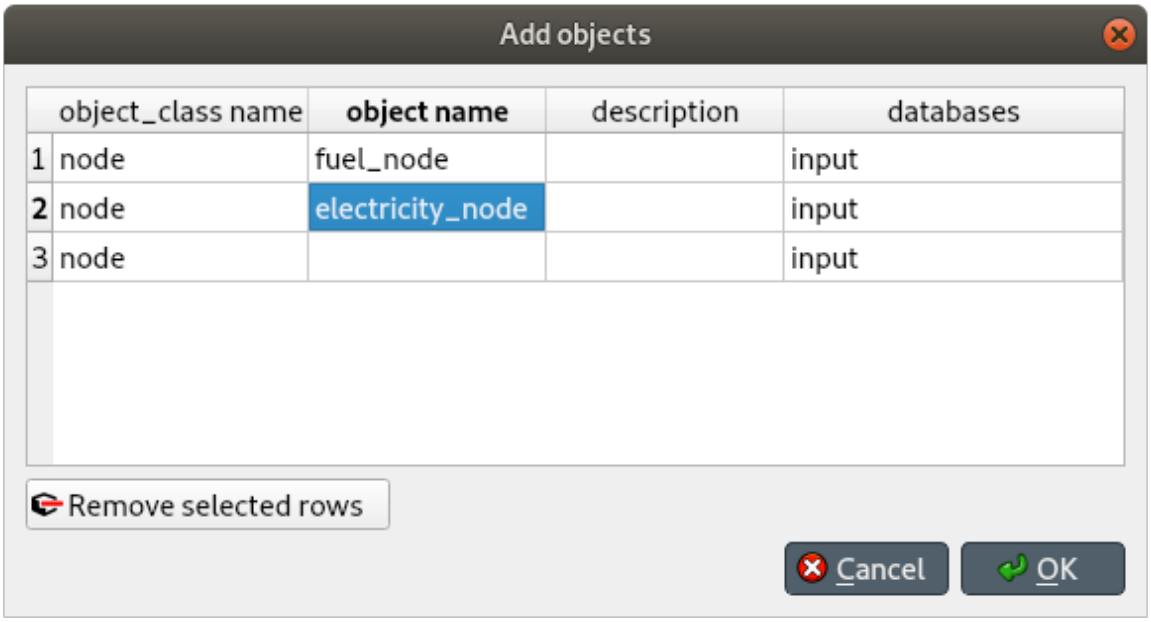
4. Press **Alt + F** to display the main menu, select **File -> Import…**, and then select the template file you previously downloaded (*spineopt_template.json*). The contents of that file will be imported into the current database, and you should then see classes like 'commodity', 'connection' and 'model' under the root node in the *Object tree* (on the left). Then import the second file (*basic_model_template.json*).

5. From the main menu, select **Session -> Commit**. Enter 'Import SpineOpt template' as message in the popup dialog, and click **Commit**.

Note: The SpineOpt basic template contains (i) the fundamental entity classes and parameter definitions that SpineOpt recognizes and expects; and (ii) some predefined entities for a common deterministic model with a 'flat' temporal structure.

Creating objects

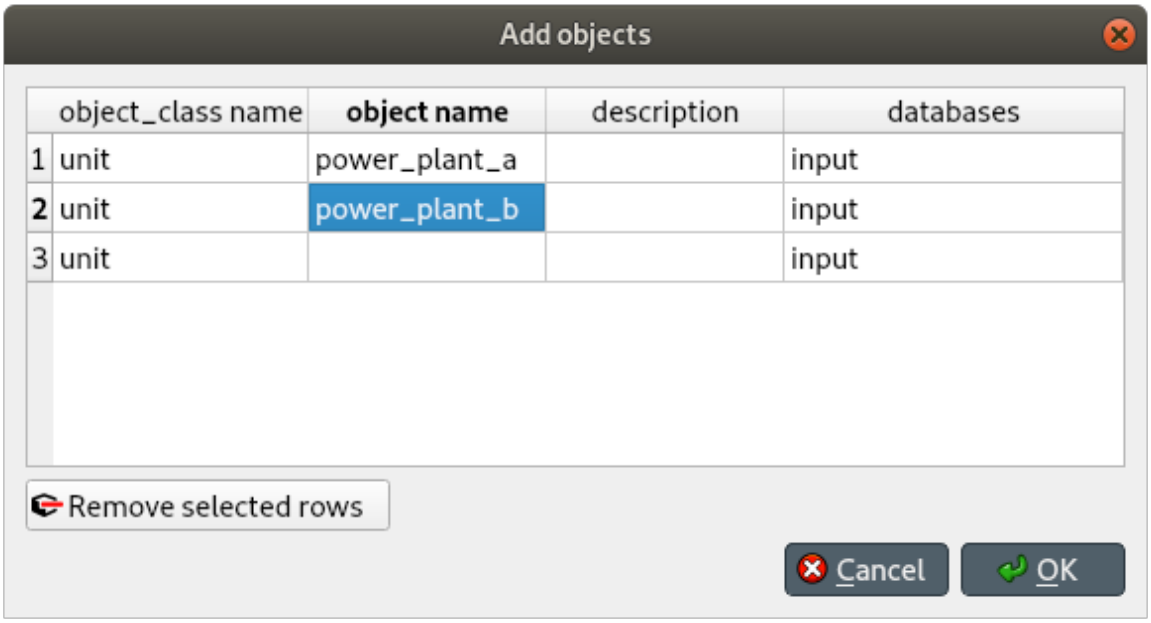
1. Always in the Spine DB editor, locate the *Object tree* (typically at the top-left). Expand the *root* element if not expanded.
2. Right click on the *node* class, and select *Add objects* from the context menu. The *Add objects* dialog will pop up.
3. Enter the names for the system nodes as seen in the image below, then press *Ok*. This will create two objects of class *node*, called *fuel_node* and *electricity_node*.



4. Right click on the *unit* class, and select *Add objects* from the context menu. The *Add objects* dialog will pop up.

Note: In SpineOpt, nodes are points where an energy balance takes place, whereas units are energy conversion devices that can take energy from nodes, and release energy to nodes.

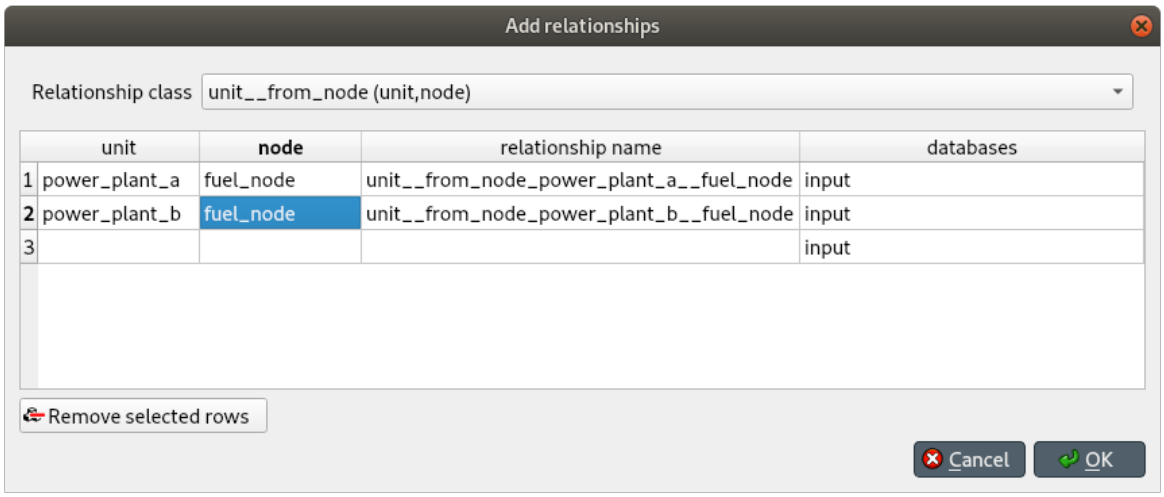
1. Enter the names for the system units as seen in the image below, then press *Ok*. This will create two objects of class *unit*, called *power_plant_a* and *power_plant_b*.



Note: To modify an object after you enter it, right click on it and select **Edit…** from the context menu.

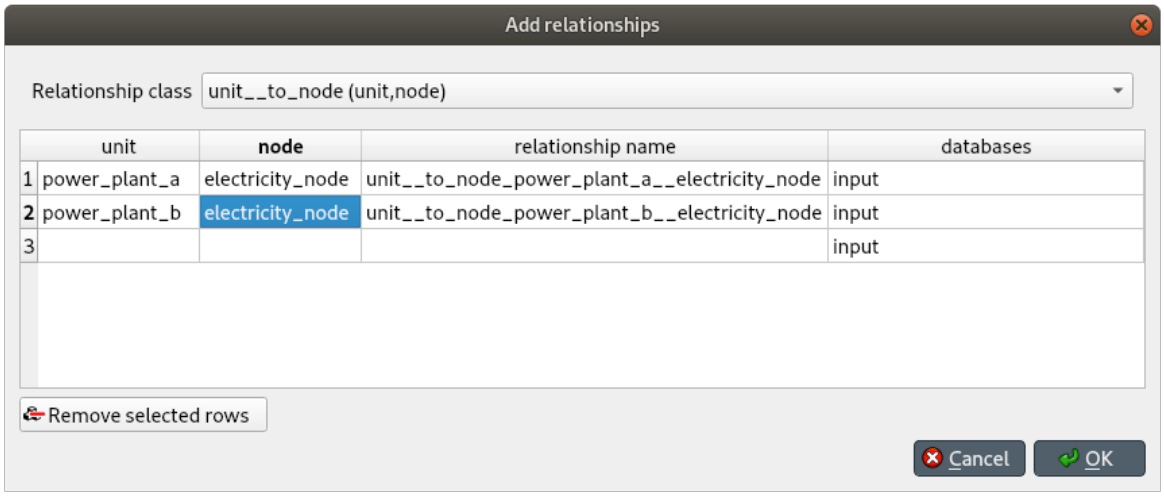
Establishing relationships

1. Always in the Spine DB editor, locate the *Relationship tree* (typically at the bottom-left). Expand the *root* element if not expanded.
2. Right click on the *unit__from_node* class, and select *Add relationships* from the context menu. The *Add relationships* dialog will pop up.
3. Select the names of the two units and their **sending** nodes, as seen in the image below; then press *Ok*. This will establish that both *power_plant_a* and *power_plant_b* take energy from the *fuel_node*.

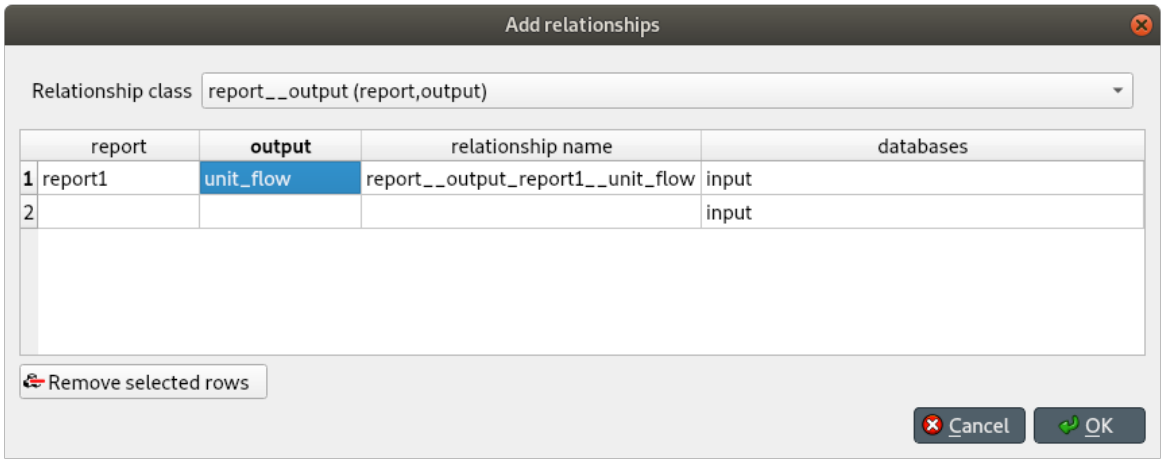


4. Right click on the *unit__to_node* class, and select *Add relationships* from the context menu. The *Add relationships* dialog will pop up.

5. Select the names of the two units and their **receiving** nodes, as seen in the image below; then press *Ok*. This will establish that both *power_plant_a* and *power_plant_b* release energy into the *electricity_node*.



6. Right click on the *report__output* class, and select *Add relationships* from the context menu. The *Add relationships* dialog will pop up.
7. Enter *report1* under *report*, and *unit_flow* under *output*, as seen in the image below; then press *Ok*. This will tell SpineOpt to write the value of the *unit_flow* optimization variable to the output database, as part of *report1*.



Note: In SpineOpt, outputs represent optimization variables that can be written to the output database as part of a report.

Specifying object parameter values

1. Back to *Object tree*, expand the *node* class and select *electricity_node*.
2. Locate the *Object parameter* table (typically at the top-center).
3. In the *Object parameter* table (typically at the top-center), select the *demand* parameter and the *Base* alternative, and enter the value *100* as seen in the image below. This will establish that there's a demand of '100' at the electricity node.

Object parameter value

object_class_name	object_name	parameter_name	alternative_name	value	database
node	electricity_node	demand	Base	150.0	input
node	electricity_node				input

4. Select *fuel_node* in the *Object tree*.
5. In the *Object parameter* table, select the *balance_type* parameter and the *Base* alternative, and enter the value *balance_type_none* as seen in the image below. This will establish that the fuel node is not balanced, and thus provide as much fuel as needed.

Object parameter value

object_class_name	object_name	parameter_name	alternative_name	value	database
node	fuel_node	balance_type	Base	balance_type_none	input
node	fuel_node				input

Specifying relationship parameter values

1. In *Relationship tree*, expand the *unit__from_node* class and select *power_plant_a | fuel_node*.
2. In the *Relationship parameter* table (typically at the bottom-center), select the *vom_cost* parameter and the *Base* alternative, and enter the value *25* as seen in the image below. This will set the operating cost for *power_plant_a*.

Relationship parameter value

relationship_class_name	object_name_list	parameter_name	alternative_name	value	database
unit__from_node	power_plant_a | fuel_node	vom_cost	Base	25.0	input
unit__from_node	power_plant_a,fuel_node				input

3. Select *power_plant_b | fuel_node* in the *Relationship tree*.
4. In the *Relationship parameter* table, select the *vom_cost* parameter and the *Base* alternative, and enter the value *50* as seen in the image below. This will set the operating cost for *power_plant_b*.

Relationship parameter value

relationship_class_name	object_name_list	parameter_name	alternative_name	value	database
unit__from_node	power_plant_b | fuel_node	vom_cost	Base	50.0	input
unit__from_node	power_plant_b,fuel_node				input

5. In *Relationship tree*, expand the *unit__to_node* class and select *power_plant_a | electricity_node*.
6. In the *Relationship parameter* table, select the *unit_capacity* parameter and the *Base* alternative, and enter the value *100* as seen in the image below. This will set the capacity for *power_plant_a*.

Relationship parameter value

relationship_class_name	object_name_list	parameter_name	alternative_name	value	database
unit__to_node	power_plant_a | electricity_node	unit_capacity	Base	100.0	input
unit__to_node	power_plant_a,electricity_node				input

7. Select *power_plant_b | electricity_node* in the *Relationship tree*.
8. In the *Relationship parameter* table, select the *unit_capacity* parameter and the *Base* alternative, and enter the value *200* as seen in the image below. This will set the capacity for *power_plant_b*.

Relationship parameter value

relationship_class_name	object_name_list	parameter_name	alternative_name	value	database
unit__to_node	power_plant_b | electricity_node	unit_capacity	Base	200.0	input
unit__to_node	power_plant_b,electricity_node				input

9. In *Relationship tree*, select the *unit__node__node* class, and come back to the *Relationship parameter* table.
10. In the *Relationship parameter* table, select *power_plant_a | electricity_node | fuel_node* under *object name list*, *fix_ratio_out_in_unit_flow* under *parameter name*, *Base* under *alternative name*, and enter *0.7* under *value*. Repeat the operation for *power_plant_b*, but this time enter *0.8* under *value*. This will set the conversion ratio from fuel to electricity for *power_plant_a* and *power_plant_b* to *0.7* and *0.8*, respectively. It should like the image below.

Relationship parameter value

relationship_class_name	object_name_list	parameter_name	alternative_name	value	database
unit__node__node	power_plant_a | electricity_node | fuel_node	fix_ratio_out_in_unit_flow	Base	0.7	input
unit__node__node	power_plant_b | electricity_node | fuel_node	fix_ratio_out_in_unit_flow	Base	0.8	input
unit__node__node					input

When you're ready, commit all changes to the database.

Executing the workflow

1. Go back to Spine Toolbox's main window, and hit the **Execute project** button from the tool bar.

   You should see 'Executing All Directed Acyclic Graphs' printed in the *Event log* (at the bottom left by default).

2. Select the 'Run SpineOpt 1' Tool. You should see the output from SpineOpt in the *Julia Console*.

Examining the results

1. Select the output data store and open the Spine DB editor.
2. Press **Alt + F** to display the main menu, and select **Pivot -> Index**.
3. Select *report__unit__node__direction__stochastic_scenario* under **Relationship tree**, and the first cell under **alternative** in the *Frozen table*.
4. Under alternative in the Frozen table, you can choose results from different runs. Pick the run you want to view. If the workflow has been run several times, the most recent run will usually be found at the bottom.
5. The *Pivot table* will be populated with results from the SpineOpt run. It will look something like the image below.

Pivot table

					(X)	
					parameter	unit_flow
report	unit	node	direction	stochastic_scenario	index	
report1	power_plant_a	electricity_node	to_node	realization	2000-01-01T00:00:00	100.0
report1	power_plant_a	fuel_node	from_node	realization	2000-01-01T00:00:00	142.857142...
report1	power_plant_b	electricity_node	to_node	realization	2000-01-01T00:00:00	50.0
report1	power_plant_b	fuel_node	from_node	realization	2000-01-01T00:00:00	62.5

3.2 Hydro Power Planning

Welcome to this Spine Toolbox tutorial for building hydro power planning models. The tutorial guides you through the implementation of different ways of modelling hydrodologically-coupled hydropower systems.

3.2.1 Introduction

This tutorial aims at demonstrating how we can model a hydropower system in Spine (*SpineOpt.jl* and *Spine-Toolbox*) with different assumptions and goals. It starts off by setting up a simple model of system of two hydropower plants and gradually introduces additional features. The goal of the model is to capture the combined operation of two hydropower plants (Språnget and Fallet) that operate on the same river as shown in the picture bellow. Each power plant has its own reservoir and generates electricity by discharging water. The plants might need to spill water, i.e., release water from their reservoirs without generating electricity, for various reasons. The water discharged or spilled by the upstream power plant follows the river route and becomes available to the downstream power plant.

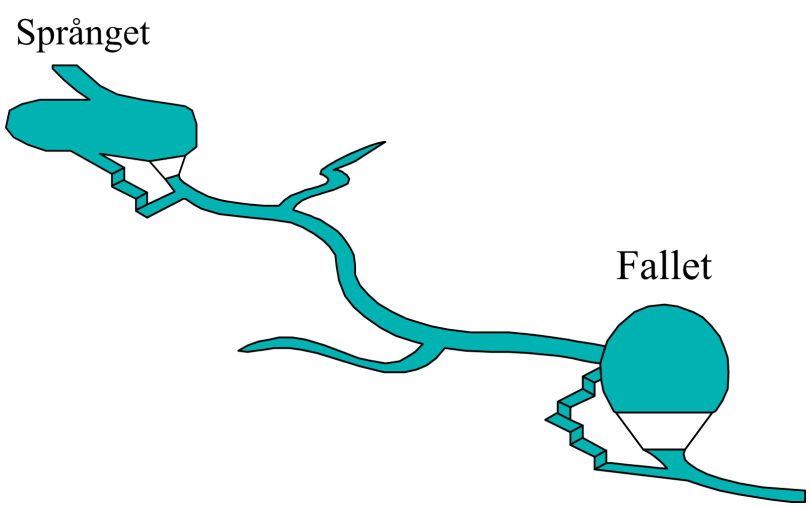


Fig. 1: A system of two hydropower plants.

In order to run this tutorial you must first execute some preliminary steps from the Simple System tutorial. Specifically, execute all steps from the guide, up to and including the step of importing-the-spineopt-database-template. It is advisable to go through the whole tutorial in order to familiarise yourself with Spine.

Note: Just remember to give a different name for the Spine Project of the hydropower tutorial (e.g., 'Two_hydro') in the corresponding step, so to not mix up the Spine Toolbox projects!

That is all you need at the moment, you can now start inserting the data.

3.2.2 Setting up a Basic Hydropower Model

For creating a SpineOpt model you need to create *Objects*, *Relationships* (associating the objects), and in some cases, parameters values accompanying them. To do this, open the input database using the Spine DB Editor (double click on the input database in the *Design View* pane of Spine Toolbox).

Note: To save your work in the Spine DB Editor you need to *commit* your changes (please check the Simple System tutorial for how to do that). As a good practice, you should commit often as you enter the data in the model to avoid data loss.

Defining objects

Commodities

Since we are modelling a hydropower system we will have to define two commodities, water and electricity. In the Spine DB editor, locate the *Object tree*, expand the root element if required, right click on the commodity class, and select *Add objects* from the context menu. In the *Add objects* dialogue that should pop up, enter the object names for the commodities as you see in the image below and then press Ok.

Nodes

Follow a similar path to add nodes, right click on the node class, and select *Add objects* from the context menu. In the dialogue, enter the node names as shown:

Nodes in SpineOpt are used to balance commodities. As you noticed, we defined two nodes for each hydropower station (water nodes) and a single electricity node. This is one possible way to model the hydropower plant operation. This will become clearer in the next steps, but in a nutshell, the *upper* node represents the water arriving at each plant, while the *lower* node represents the water that is discharged and becomes available to the next plant.

Connections

Similarly, add connections, right click on the connection class, select *Add objects* from the context menu and add the following connections:

Connections enable the nodes to interact. Since, for each plant we need to model the amount of water that is discharged and the amount that is spilled, we must define two connections accordingly. When defining relationships we shall associate the connections with the nodes.

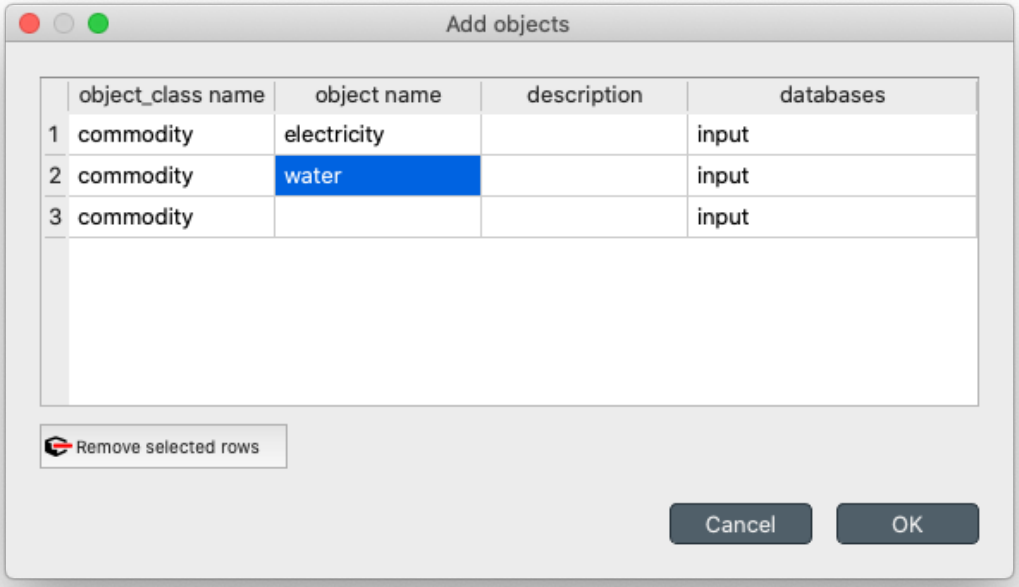


Fig. 2: Defining commodities.

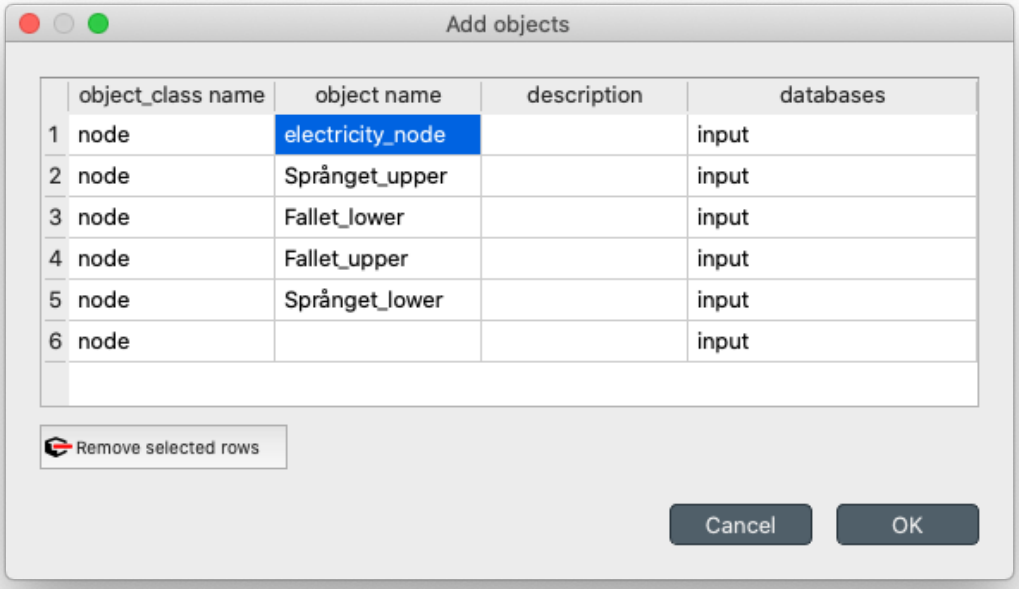


Fig. 3: Defining nodes.

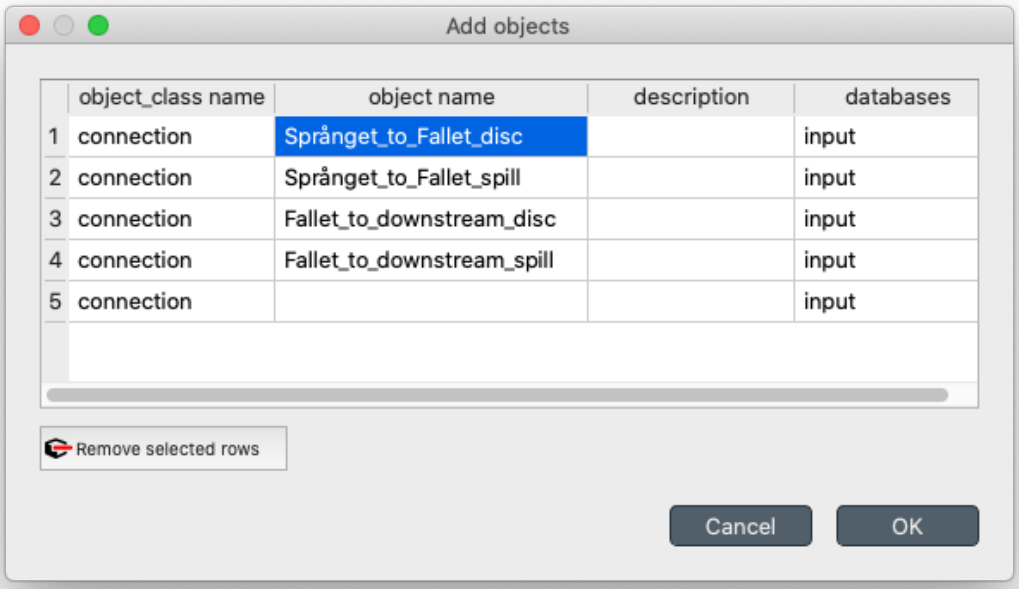


Fig. 4: Defining connections.

Units

To convert from one type of commodity associated with one node to another, you need a unit. You guessed it! Right click on the unit class, select *Add objects* from the context menu and add the following units:

We have defined one unit for each hydropower plant that converts water to electricity and an additional unit that we will use to model the income from selling the electricity production in the electricity market.

Relationships

Assinging commodities to nodes

Since we have defined more than one commodities, we need to assign them to nodes. In the Spine DB editor, locate the *Relationship tree*, expand the root element if required, right click on the *node__commodity* class, and select *Add relationships* from the context menu. In the *Add relationships* dialogue, enter the following relationships as you see in the image below and then press Ok.

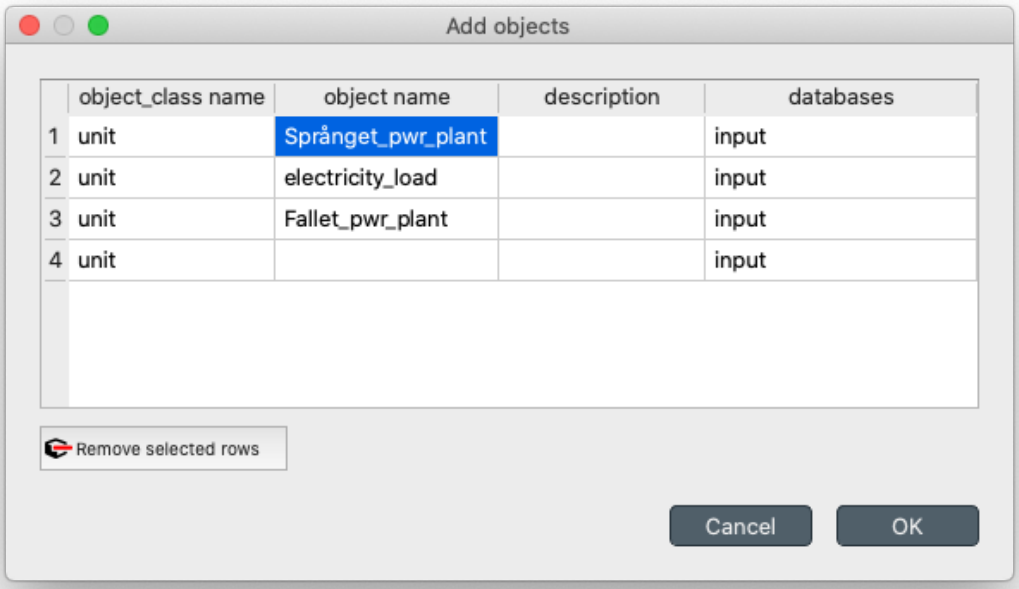


Fig. 5: Defining units.

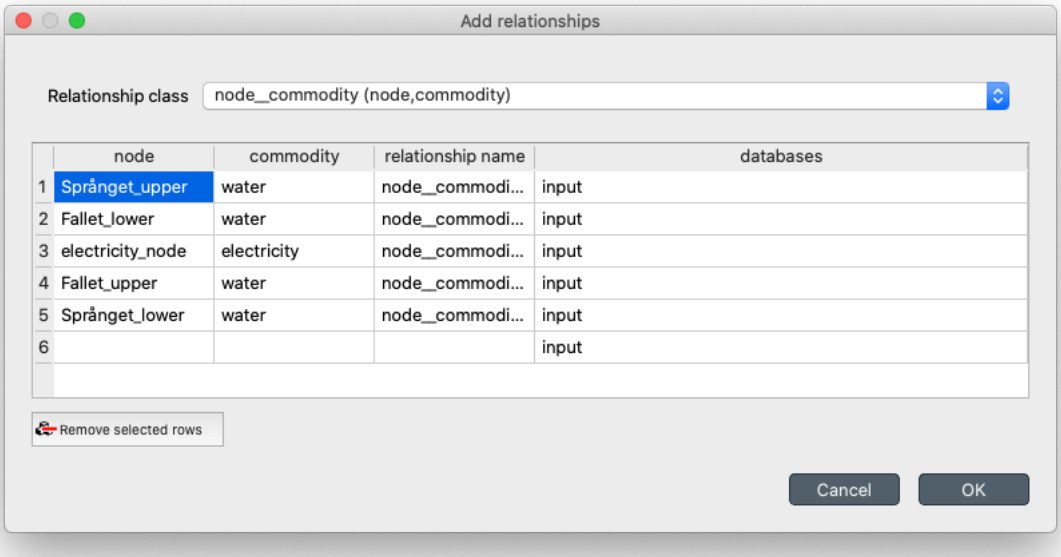


Fig. 6: Introducing *node__commodity* relationships.

Associating connections to nodes

Next step is to define the topology of flows between the nodes. To do that insert the following relationships in the *connection__from_node* class:

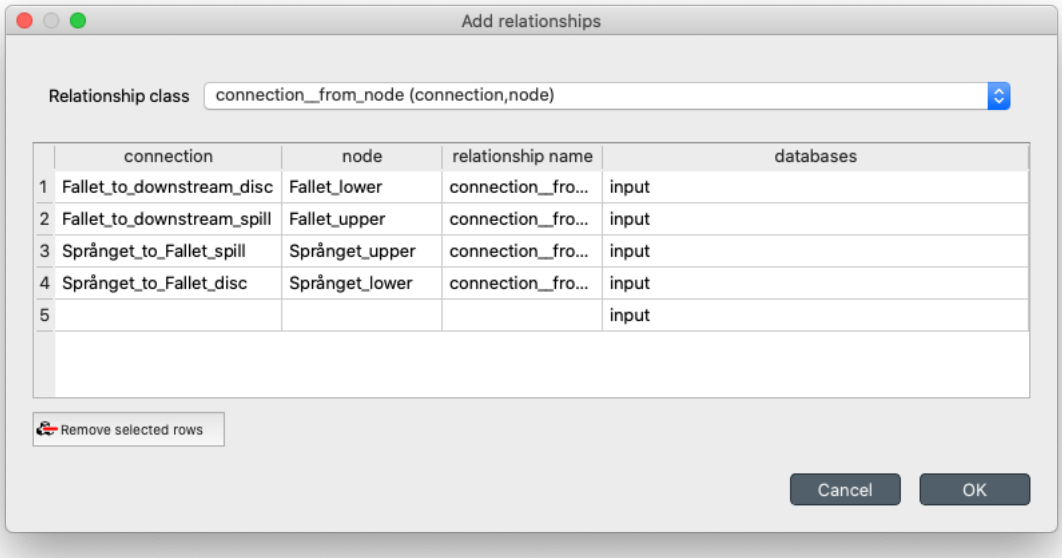


Fig. 7: Introducing *connection__from_node* relationships.

as well as the following the following *connection__node_node* relationships as you see in the figure:

Placing the units in the model

To define the topology of the units and be able to introduce their parameters later on, you need to define the following relationships in the *unit__from_node* class:

in the *unit__node_node* class:

and in the *unit__to_node* class as you see in the following figure:

Defining the report outputs

To force Spine to export the optimal values of the optimization variables to the output database you need to specify them in the form of *report_output* relationships. Add the following relationships to the *report_output* class:

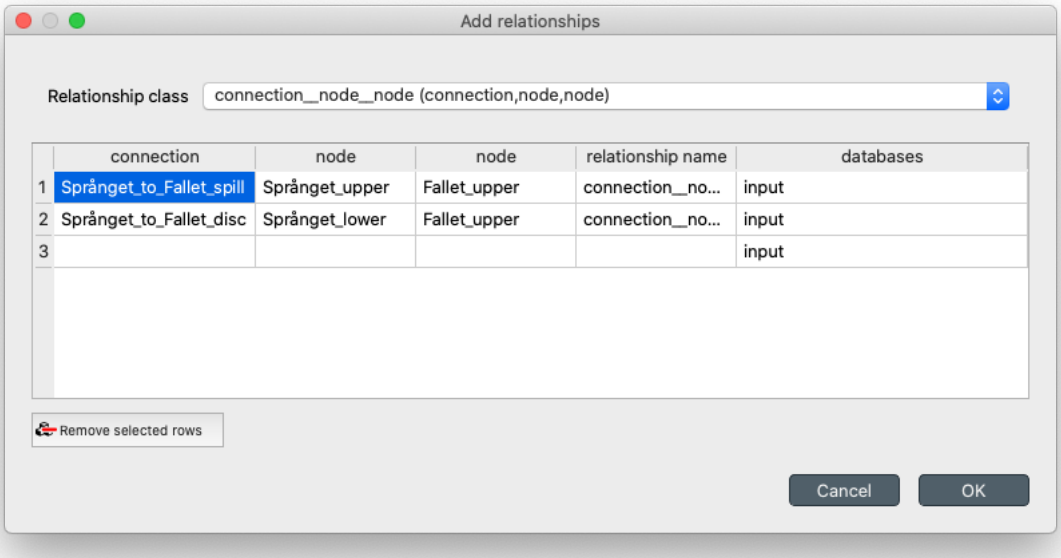


Fig. 8: Introducing *connection__node_node* relationships.

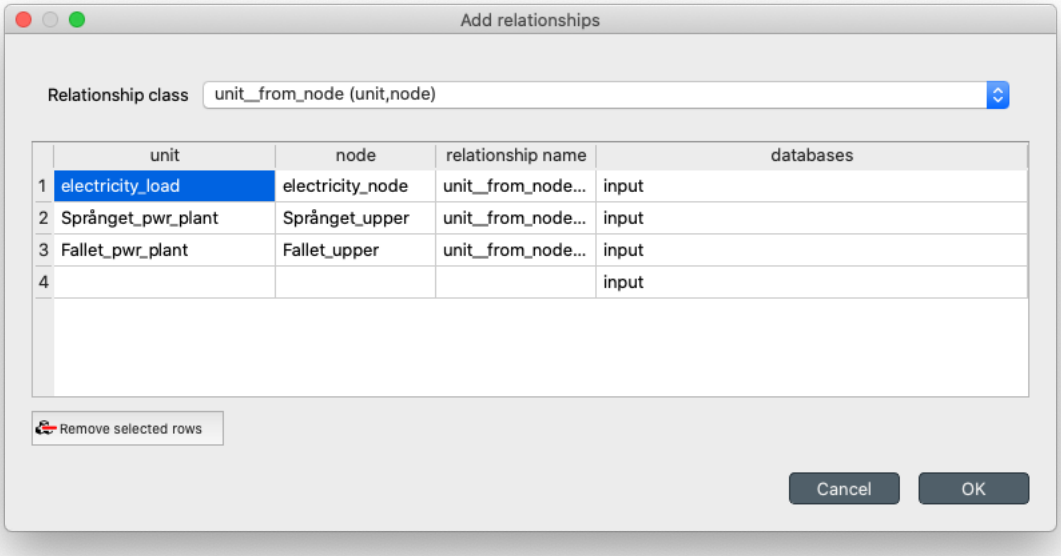


Fig. 9: Introducing *unit__from_node* relationships.

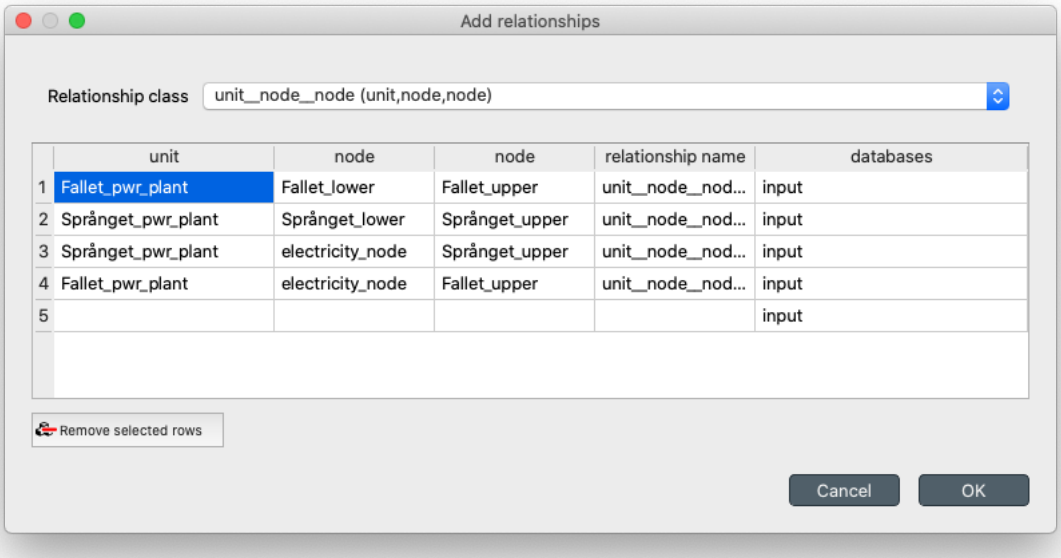


Fig. 10: Introducing *unit__node_node* relationships.

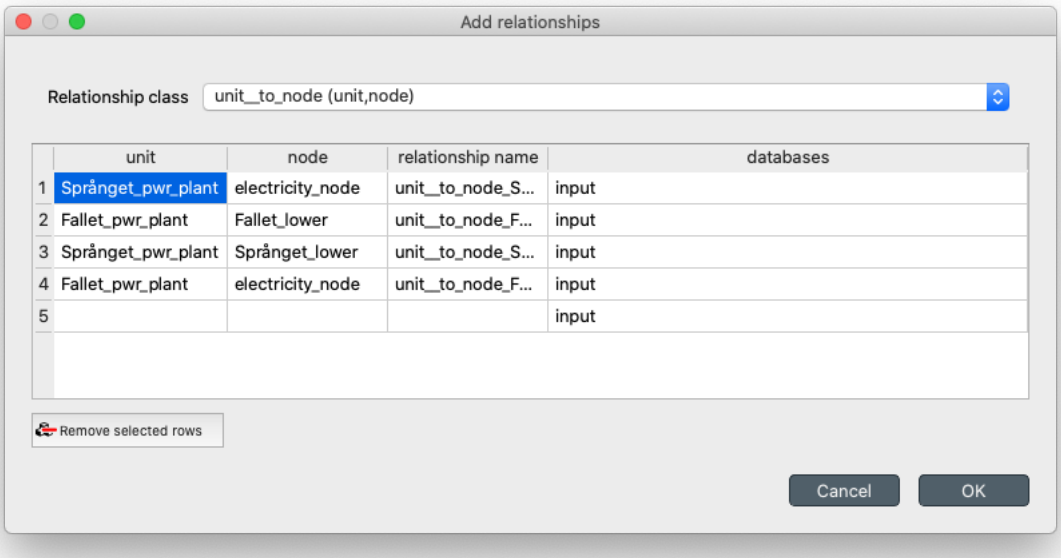


Fig. 11: Introducing *unit__to_node* relationships.

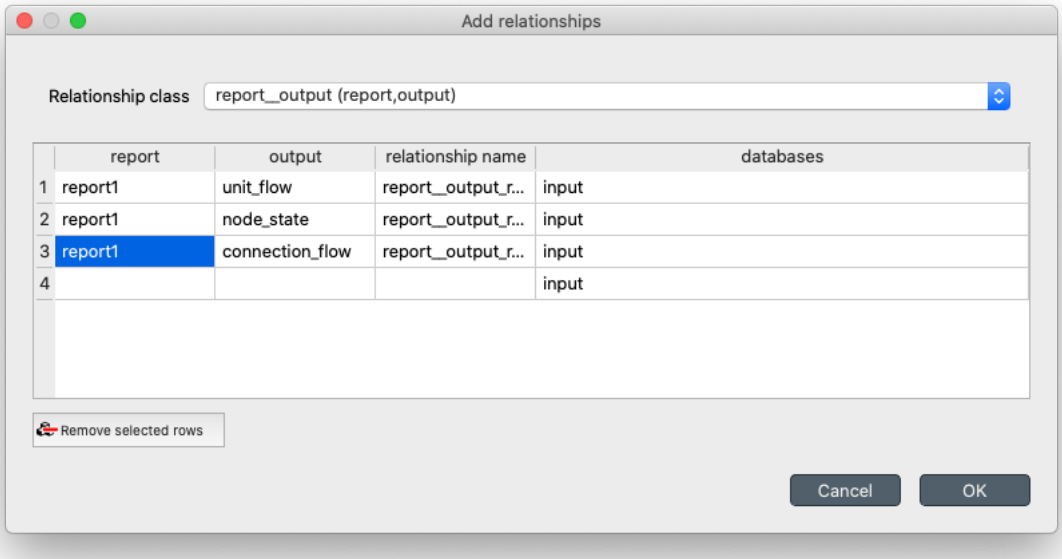


Fig. 12: Introducing report outputs with *report_output* relationships.

Objects and Relationships parameter values

Defining model parameter values

The specify modelling properties of both objects and relationships you need to introduce respective parameter values. To introduce object parameter values first select the *model* class in the Object tree and enter the following values in the *Object parameter value* pane:

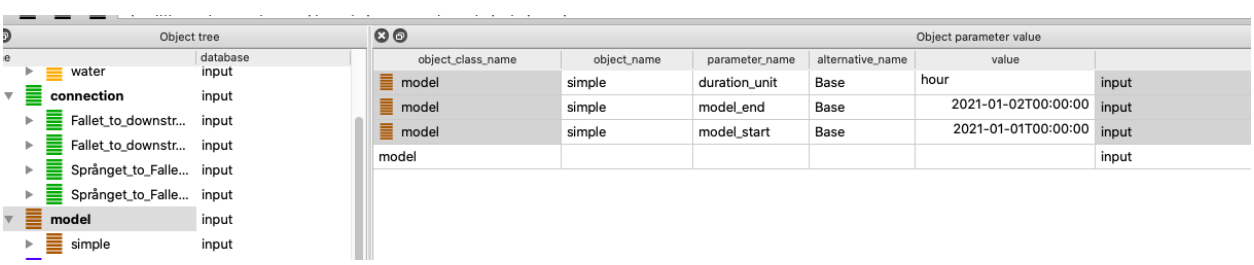
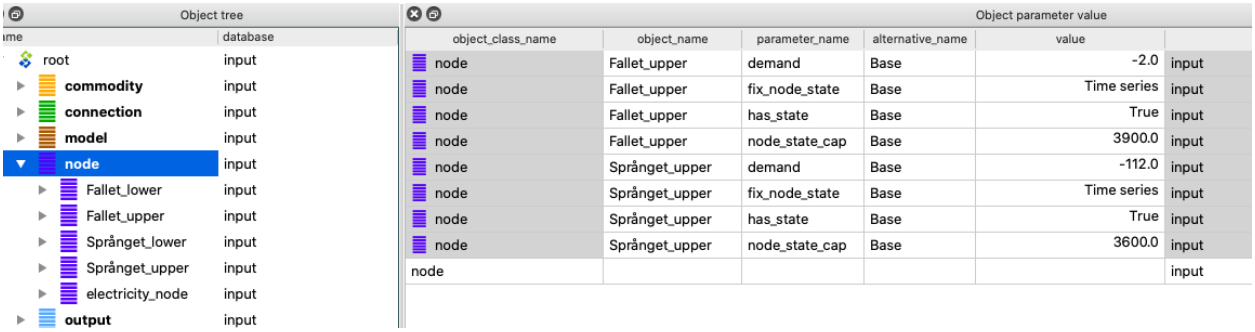


Fig. 13: Defining model execution parameters.

Observe the difference between the *Object parameter value* and the *Object parameter definition* sub-panes of the *Object parameter value* pane. The first one is for the modeller to introduce values for specific parameters, while the second one holds the definition of all available parameters with their default values (these are overwritten when the user introduces their own values). Feel free to explore the different parameters and their default values. While entering data in each row you will also observe that, in most cases, clicking on each cell activates a drop-down list of elements that the user must choose from. In the case of the *value* cells, however, unless you need to input a scalar value or a string, you should right-click on the cell and select edit for specifying the data type of the parameter value. As you see in the figure above, for the first *duration_unit* parameter you is of type string, while the *model_start* and *model_end* parameters are of type Date time. The Date time parameters can be edited by right-clicking on the corresponding *value* cells, selecting *Edit*, and then inserting the Date time values that you see in the figure above in the *Datetime* field using the correct format.

Defining node parameter values

Going back to hydropower modelling, we need to specify several parameters for the nodes of the systems. In the same pane as before, but this time selecting the *node* class from the *Object tree*, we need to add the following entries:



Object tree	database
root	input
commodity	input
connection	input
model	input
node	input
Fallet_lower	input
Fallet_upper	input
Språnget_lower	input
Språnget_upper	input
electricity_node	input
output	input

object_class_name	object_name	parameter_name	alternative_name	value	
node	Fallet_upper	demand	Base	-2.0	input
node	Fallet_upper	fix_node_state	Base	Time series	input
node	Fallet_upper	has_state	Base	True	input
node	Fallet_upper	node_state_cap	Base	3900.0	input
node	Språnget_upper	demand	Base	-112.0	input
node	Språnget_upper	fix_node_state	Base	Time series	input
node	Språnget_upper	has_state	Base	True	input
node	Språnget_upper	node_state_cap	Base	3600.0	input
node					input

Fig. 14: Defining model execution parameters.

Before we go through the interpretation of each parameter, click on the following link for each *fix_node_state* parameter (Node state Språnget, Node state Fallet), select all, copy the data and then paste them directly in the respective parameter value cell. Spine should automatically detect and input the timeseries data as a parameter value. The data type for those entries should be *Timeseries* as shown in the figure above. Alternatively, you can select the data type as *Timeseries* and manually insert the data (values with their corresponding datetimes).

To model the reservoirs of each hydropower plant, we leverage the *state* feature that a node can have to represent storage capability. We only need to do this for one of the two nodes that we have used to model each plant and we choose the *upper* level node. To define storage, we set the value of the parameter *has_state* as True (be careful to not set it as a string but select the boolean true value by right clicking and selecting Edit in the respective cells). This activates the storage capability of the node. Then, we need to set the capacity of the reservoir by setting the *node_state_cap* parameter value. Finally, we fix the initial and final values of the reservoir by setting the parameter *fix_node_state* to the respective values (we introduce *nan* values for the time steps that we don't want to impose such constraints). To model the local inflow we use the *demand* parameter but using the negated value of the actual inflow, due to the definition of the parameter in Spine as a **demand**.

Defining the temporal resolution of the model

Spine automates the creation of the temporal resolution of the optimization model and even supports different temporal resolutions for different parts of the model. To define a model with an hourly resolution we select the *temporal_block* class in the *Object tree* and we set the *resolution* parameter value to *1h* as shown in the figure:

Defining connection parameter values

The water that is discharged from Språnget will flow from *Språnget_lower* node to *Fallet_upper* through the *Språnget_to_Fallet_disc* connection, while the water that is spilled will flow from *Språnget_upper* directly to to *Fallet_upper* through the *Språnget_to_Fallet_spill* connection. To model this we need to select the *connection__node_node* class in the Relationship tree and add the following entries in the *Relationship parameter value* pane, as shown next:

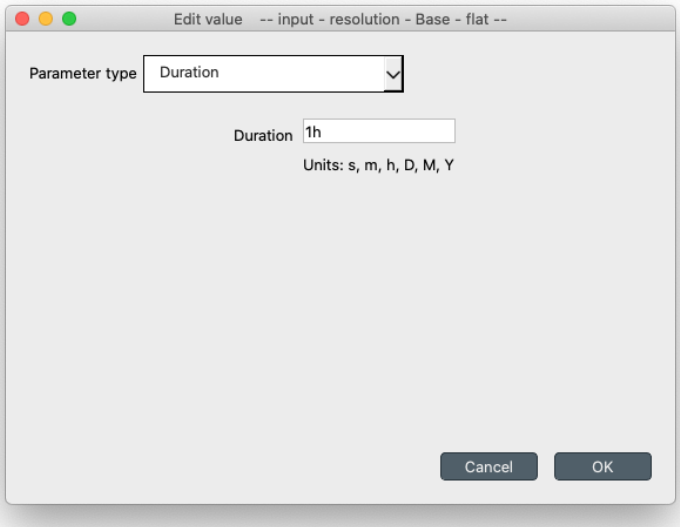


Fig. 15: Setting the temporal resolution of the model.

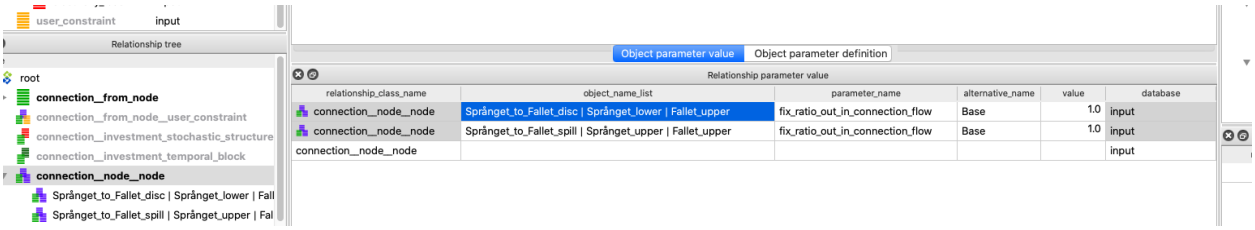


Fig. 16: Defining discharge and spillage ratio flows.

Defining unit parameter values

Similarly, for each one of the *unit__from_node*, *unit__node_node*, and *unit__to_node* relationship classes we need to add the the maximal water that can be discharged by each hydropower plant:

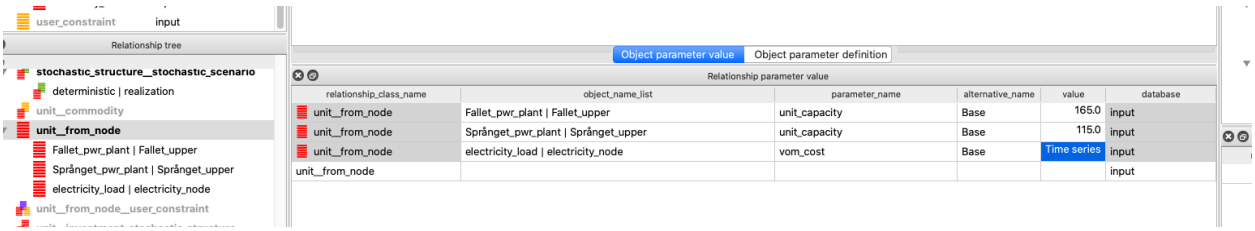


Fig. 17: Setting the maximal water discharge of each plant.

To define the income from selling the produced electricity we use the *vom_cost* parameter and negate the values of the electricity prices. To automatically insert the timeseries data in Spine, click on the Electricity prices timeseries, select all values, copy, and paste them, after having selected the value cell of the corresponding row. You can plot and edit the timeseries data by double clicking on the same cell afterwards:

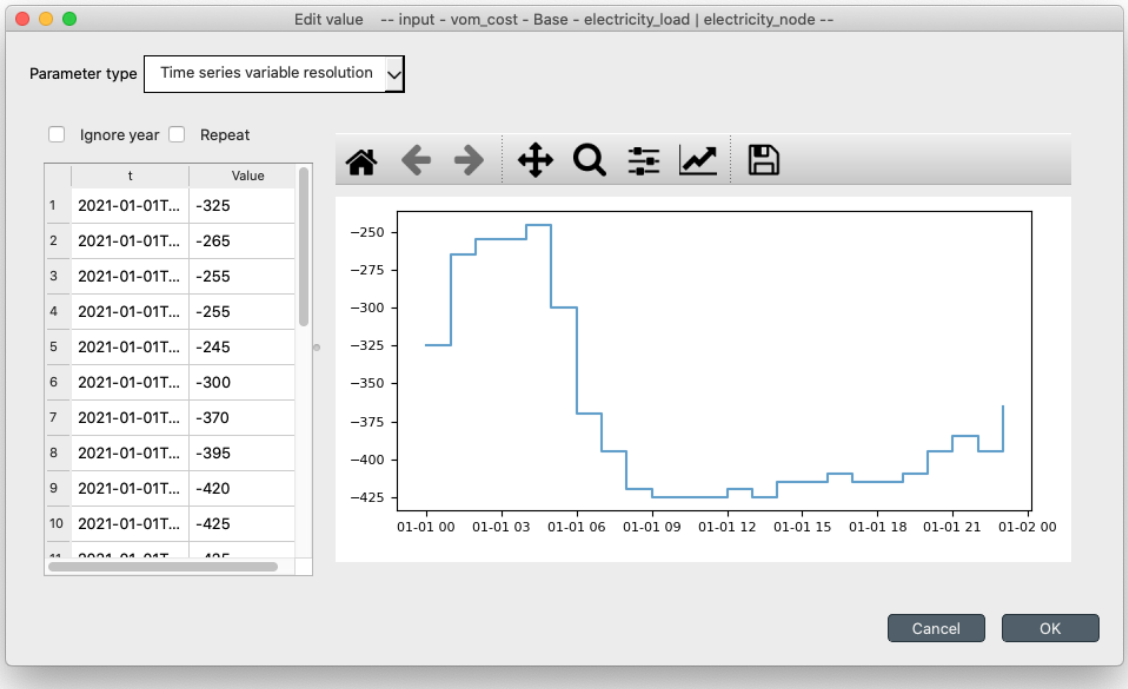


Fig. 18: Previewing and editing the electricity prices timeseries.

Carrying on with our hydropower model we must define the conversion ratios between the nodes. Assuming that water is not "lost" from the *upper* node toward the *lower* node and electricity is produced with the discharged water with a given efficiency we define the following parameter values for each hydropower plant, in the *unit__node_node* class:

Lastly, we can define the maximal electricity production of each plant by inserting the following *unit__to_node* relationship parameter values:

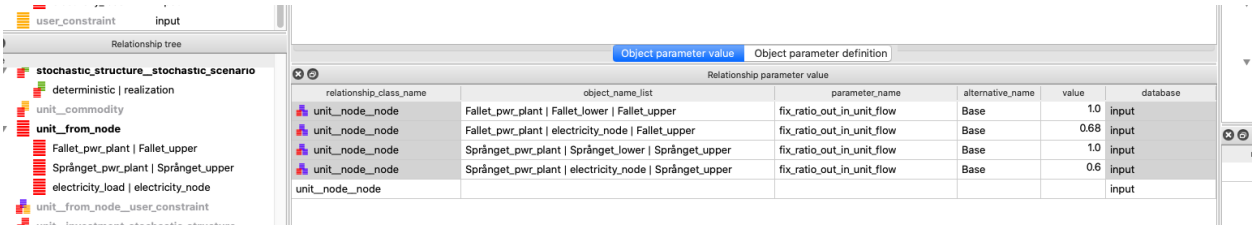


Fig. 19: Defining conversion efficiencies.

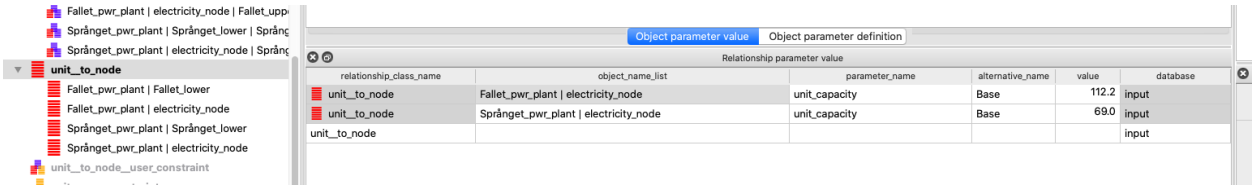


Fig. 20: Setting the maximal electricity production of each plant.

Hooray! You can now commit the database, close the Spine DB Editor and run your model! Go to the main Spine window and click on Execute .

Examining the results

Select the output data store and open the Spine DB editor. To quickly plot some results, you can expand the unit class in the Object tree and select the *electricity_load* unit. In the *Relationship parameter value* pane double click on the value cell of

> **report1|electricity_load|electricity_node|from_node|realization**

object name. This will open a plotting window from were you can also examine closer and retrieve the data, as shown in the next figure. The *unit_flow* variable of the *electricity_load* unit represents the total electricity production in the system:

Now, take to a minute to reflect on how you could retrieve the data representing the water that is discharged by each hydropower plant as shown in the next figure:

The right answer is that you need to select some hydropower plant (e.g., Spånget) and then double-click on the value cell of the object name

> **report1|Spånget_pwr_plant|Spånget_lower|to_node|realization**, or **report1|Spånget_pwr_plant|Spånget_upper|from_node|realization**.

It could be useful to also reflect on why these objects give the same results, and what do the results from the third element represent. (Hint: observe the *to_* or *from_* directions in the object names). As an exercise, you can try to retrieve the timeseries data for spilled water as well as the water levels at the reservoir of each hydropower plant.

You can further explore the model, or make changes in the input database to observe how these affect the results, e.g., you can use different electricity prices, values for the reservoir capacity (and initialization points), as well as change the temporal resolution of the model. All you need to do is commit the changes and run your model. Every time that you run the model, your results are appended in the output database with an execution timestamp. You can however filter your results per execution, by selecting the *Alternative* that you want from the *Alternative/Scenario tree* pane. You can use the exporter too to export specific variables in an Excel sheet. Alternatively, you can export all the data of the output database by going to the main menu (Press **Alt + F** to display it), selecting **File -> Export**, then select the items that you want, click ok and export the data in Excel, or json format.

In the following, we extend this simple hydropower system to include more elaborate modelling choices.

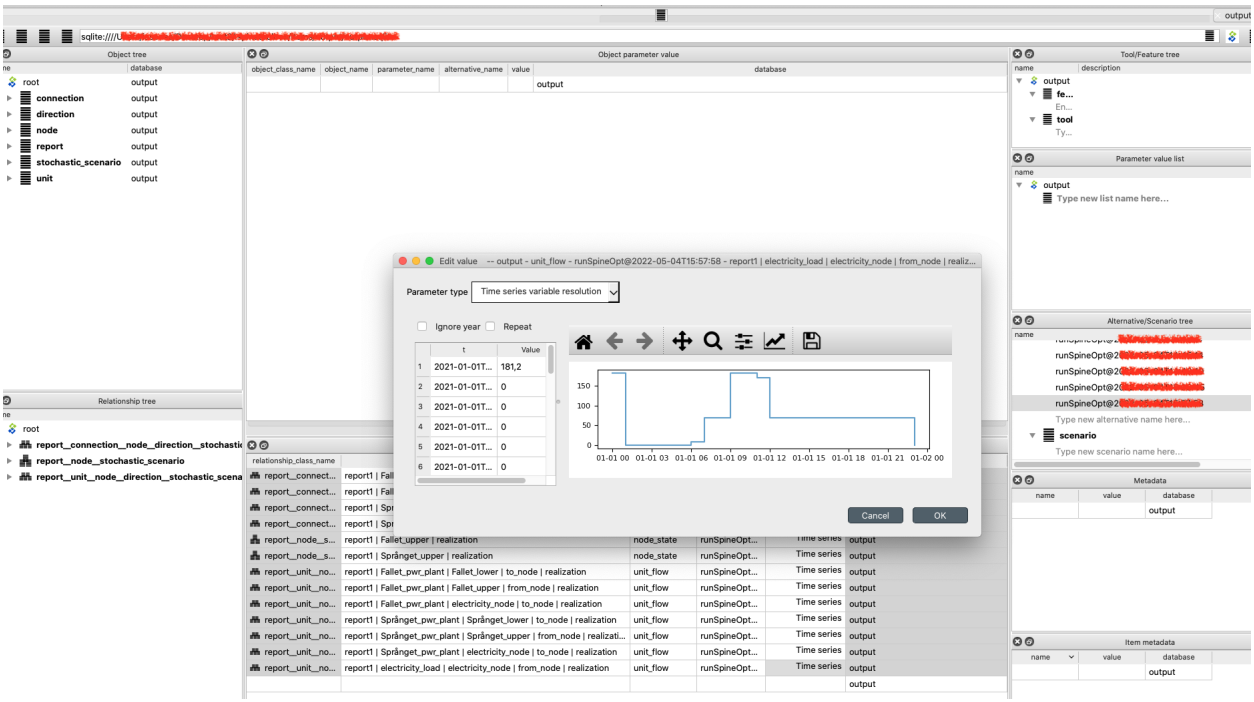


Fig. 21: Total electricity produced in the system.

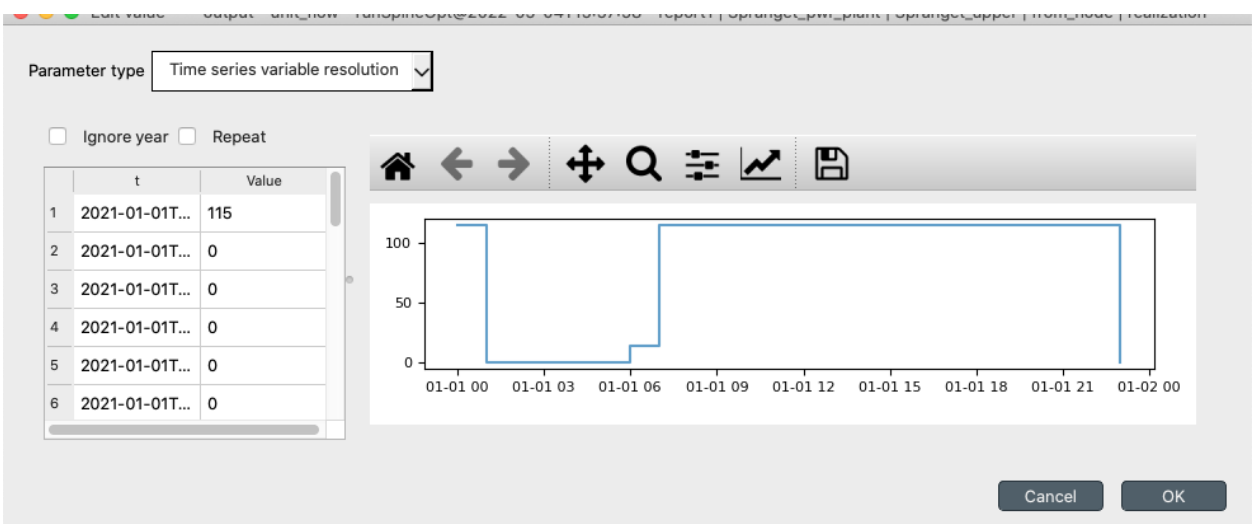


Fig. 22: Water discharge of Språnget hydropower plant.

Note: In each of the next sections, we perform incremental changes to the initial simple hydropower model. If you want to keep the database that you created, you can duplicate the database file (right-click on the input database and select **Duplicate and duplicate files**) and perform the changes in the new database. You need to configure the workflow accordingly in order to run the database you want (please check the Simple System tutorial for how to do that).

3.2.3 Maximisation of Stored Water

Instead of fixing the water content of the reservoirs at the end of the planning period, we can consider that the remaining water in the reservoirs has a value and then maximize the value along with the revenues for producing electricity within the planning horizon. This objective term is often called the **Value of stored water** and we can approximate it by assuming that this water will be used to generate electricity in the future that would be sold at a forecasted price. The water stored in the upstream hydropower plant will become also available to the downstream plant and this should be taken into account.

To model the value of stored water we need to make some additions and modifications to the initial model.

1. First, add a new node (see *adding nodes*) and give it a name (e.g., *stored_water*). This node will accumulate the water stored in the reservoirs at the end of the planning horizon. Associate the node with the water commodity (see *node__commodity*).
2. Add three more units (see *adding units*); two will transfer the water at the end of the planning horizon in the new node that we just added (e.g., *Språnget_stored_water*, *Fallet_stored_water*), and one will be used as a *sink* introducing the value of stored water in the objective function (e.g., *value_stored_water*).
3. To establish the topology of the new units and nodes (see *adding unit relationships*):
   - add one *unit__from_node* ` relationship, between the *value_stored_water* unit from the *stored_water* node,

   another one between the *Språnget_stored_water* unit from the *Språnget_upper* node and one for *Faller_stored_water* from *Fallet_upper*.
   - add one *unit__node__node* relationship between the *Språnget_stored_water* unit with the *stored_water* and *Språnget_upper* nodes and another one for *Fallet_stored_water* unit with the *stored_water* and *Fallet_upper* nodes,
   - add a *unit__to_node* relationship between the *Fallet_stored_water* and the *stored_water* node and another one between the *Språnget_stored_water* unit and the *stored_water* node.
4. Now we need to make some changes in object parameter values.
   - Extend the planning horizon of the model by one hour, i.e., change the *model_end* parameter value to *2021-01-02T01:00:00* (right-click on the value cell, click edit and paste the new datetime in the popup window).
   - Remove the *fix_node_state* parameter values for the end of the optimization horizon as you seen in the following figure: double click on the *value* cell of the *Språnget_upper* and *Fallet_upper* nodes, select the third data row, right-click, select *Remove rows*, and click OK.
   - Add an electricity price for the extra hour. Enter the parameter *vom_cost* on the *unit__from_node* relationship between the *electricity_node* and the *electricity_load* and set 0 as the price of electricity for the last hour *2021-01-02T00:00:00*. The price is set to zero to ensure no electricity is sold during this hour.
5. Finally, we need to add some relationship parameter values for the new units:
   - Add a *vom_cost* parameter value on a *value_stored_water|stored_water* instance of a *unit__from_node* relationship, as you see in the figure bellow. For the timeseries you can copy-paste the data directly from this link. If you examine the timeseries data you'll notice that we have imposed a zero cost for all the

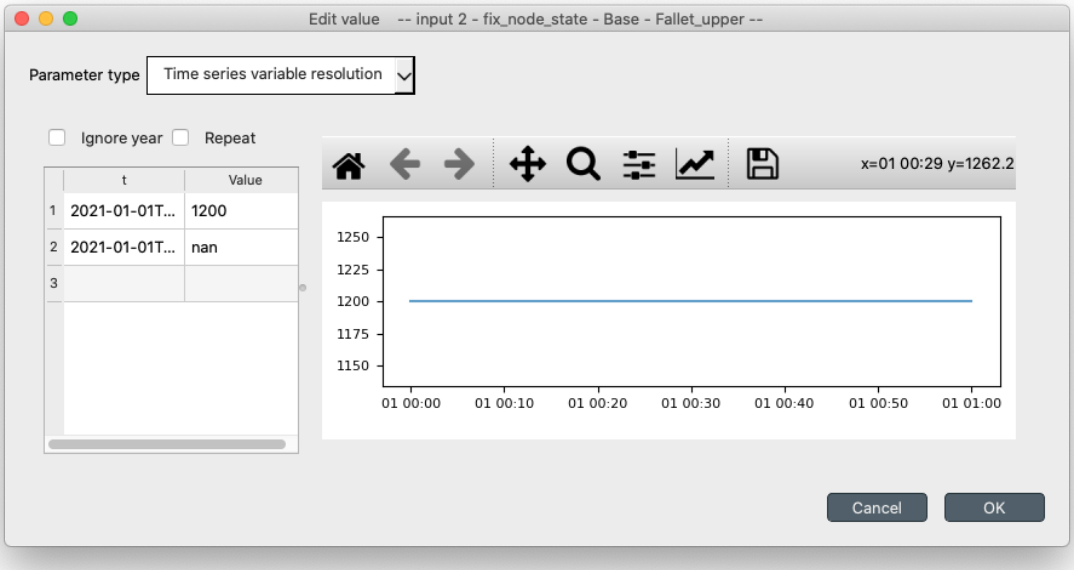


Fig. 23: Modify the *fix_node_state* parameter value of *Språnget_upper* and *Fallet_upper* nodes.

optimisation horizon, while we use an assumed future electricity value for the additional time step at the end of the horizon.

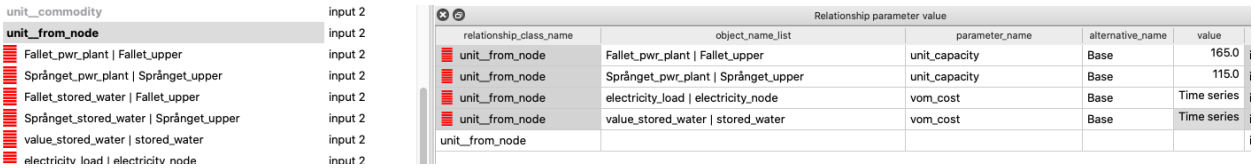


Fig. 24: Adding *vom_cost* parameter value on the *value_stored_water* unit.

- Add two *fix_ratio_out_in_unit_flow* parameter values as you see in the figure bellow. The efficiency of *Fallet_stored_water* is the same as the *Fallet_pwr_plant* as the water in Fallet's reservoir will be used to produce electricity by the the Fallet plant only. On the other hand, the water from Språnget's reservoir will be used both by Fallet and Språnget plant, therefore we use the sum of the two efficiencies in the parameter value of *Språnget_stored_water*.

You can now commit your changes in the database, execute the project and *examine the results*! As an exercise, try to retrieve the value of stored water as it is calculated by the model.

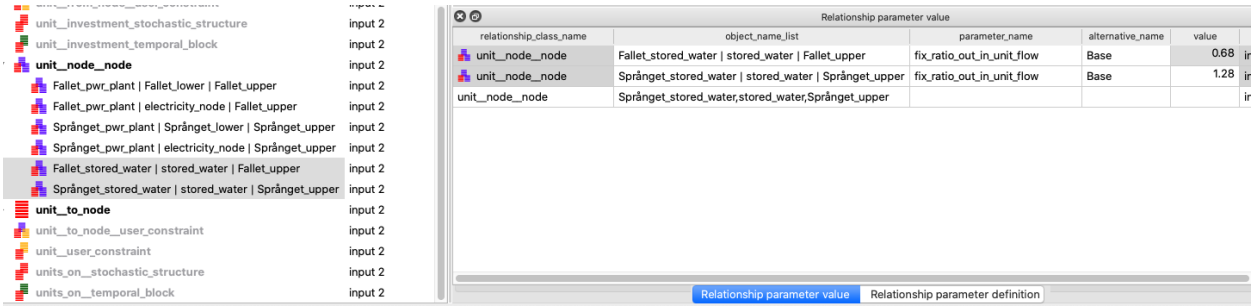


Fig. 25: Adding *fix_ratio_out_in_unit_flow* parameter values on the *Språnget_stored_water* and *Fallet_stored_water* units.

3.2.4 Spillage Constraints - Minimisation of Spilt Water

It might be the case that we need to impose certain limits to the amount of water that is spilt on each time step of the planning horizon, e.g., for environmental reasons, there can be a minimum and a maximum spillage level. At the same time, to avoid wasting water that could be used for producing electricity, we could explicitly impose the spillage minimisation to be added in the objective function.

1. Add one unit (see *adding units*) to impose the spillage constraints to each plant and name it (for example *Språnget_spill*).
2. Remove the *Språnget_to_Fallet_spill* connection (in the Object tree expand the connection class, right-click on *Språnget_to_Fallet_spill*, and the click **Remove**).
3. To establish the topology of the unit (see *adding unit relationships*):
   - Add a *unit__from_node* relationship, between the *Språnget_spill* unit from the *Språnget_upper* node,
   - add a *unit__node__node* relationship between the *Språnget_spill* unit with the *Fallet_upper* and *Språnget_upper* nodes,
   - add a *unit__to_node* relationship between the *Språnget_spill* and the *Fallet_upper* node,
4. Add the relationship parameter values for the new units:
   - Set the *unit_capacity* (to apply a maximum), the *minimum_operating_point* (defined as a percentage of the *unit_capacity*) to impose a minimum, and the *vom_cost* to penalise the water that is spilt:

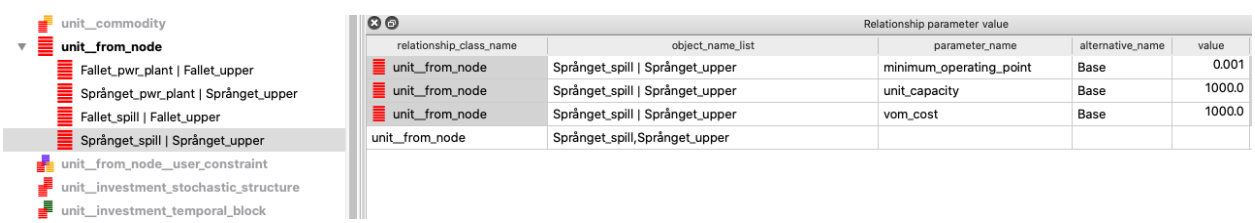


Fig. 26: Setting minimum (the minimal value is defined as percentage of capacity), maximum, and spillage penalty.

   - For the *Språnget_spill* unit define the *fix_ratio_out_in_unit_flow* parameter value of the *min_spillage|Fallet_upper|Språnget_upper* relationship to **1** (see *adding unit relationships*).

Commit your changes in the database, execute the project and *examine the results*! As an exercise, you can perform this process for and Fallet plant (you would also need to add another water node, downstream of Fallet).

3.2.5 Follow Contracted Load Curve

It is often the case that a system of hydropower plants should follow a given production profile. To model this in the given system, all we have to do is set a demand in the form of a timeseries to the *electricity_node*.

1. Add the Contracted load timeseries, to the *demand* parameter value of the *electricity_node* (see *adding node parameter values*).

Commit your changes in the database, execute the project and *examine the results*!

This concludes the tutorial, we hope that you enjoyed building hydropower systems in Spine as much as we do!

3.3 Case Study A5 tutorial

Welcome to this Spine Toolbox Case Study tutorial. Case Study A5 is one of the Spine Project case studies designed to verify Toolbox and Model capabilities. To this end, it *reproduces* an already existing study about hydropower on the Skellefte river, which models one week of operation of the fifteen power stations along the river.

This tutorial provides a step-by-step guide to run Case Study A5 on Spine Toolbox and is organized as follows:

3.3.1 Introduction

Model assumptions

For each power station in the river, the following information is known:

- The capacity, or maximal electricity output. This datum also provides the maximal water discharge as per the efficiency curve (see next point).
- The efficiency curve, or conversion rate from water to electricity. In this study, a piece-wise linear efficiency with two segments is assumed. Moreover, this curve is monotonically decreasing, i.e., the efficiency in the first segment is strictly greater than the efficiency in the second segment.
- The maximal reservoir level (maximal amount of water that can be stored in the reservoir).
- The reservoir level at the beginning of the simulation period and at the end.
- The minimal amount of water that the plant must discharge at every hour. This is usually zero (except for one of the plants).
- The downstream plant, or next plant in the river course.
- The time that it takes for the water to reach the downstream plant. This time can be different depending on whether the water is discharged (goes through the turbine) or spilled.
- The local inflow, or amount of water that naturally enters the reservoir at every hour. In this study, it is assumed constant over the entire simulation period.
- The hourly average water discharge. It is assumed that before the beginning of the simulation, this amount of water has constantly been discharged at every hour.

The system is operated so as to maximize the total profit over the planning week, while respecting capacity constraints, maximal reservoir level constrains, etc. Hourly profit per plant is simply computed as the product of the electricity price and the production.

Modelling choices

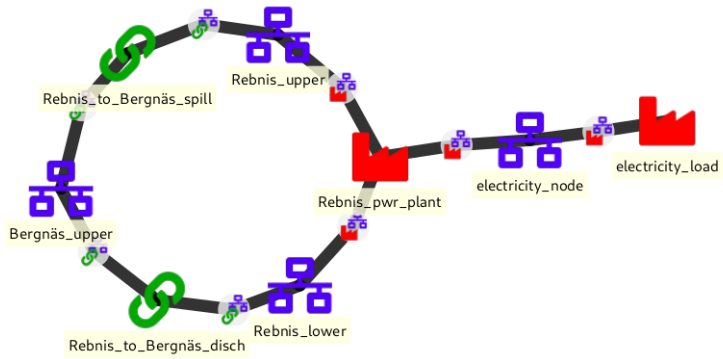
The model of the electric system is fairly simple, only two elements are needed:

- A common electricity node.
- A load unit that takes electricity from that node.

On the contrary, the model of the river system is more detailed. Each power station in the river is modelled using the following elements:

- An upper water node, located at the entrance of the station.
- A lower water node, located at the exit of the station.
- A power plant unit, that discharges water from the upper node into the lower node, and feeds electricity produced in the process to the common electricity node.
- A spillway connection, that takes spilled water from the upper node and releases it to the downstream upper node.
- A discharge connection, that takes water from the lower node and releases it to the downstream upper node.

Below is a schematic of the model. For clarity, only the Rebnis station is presented in full detail:

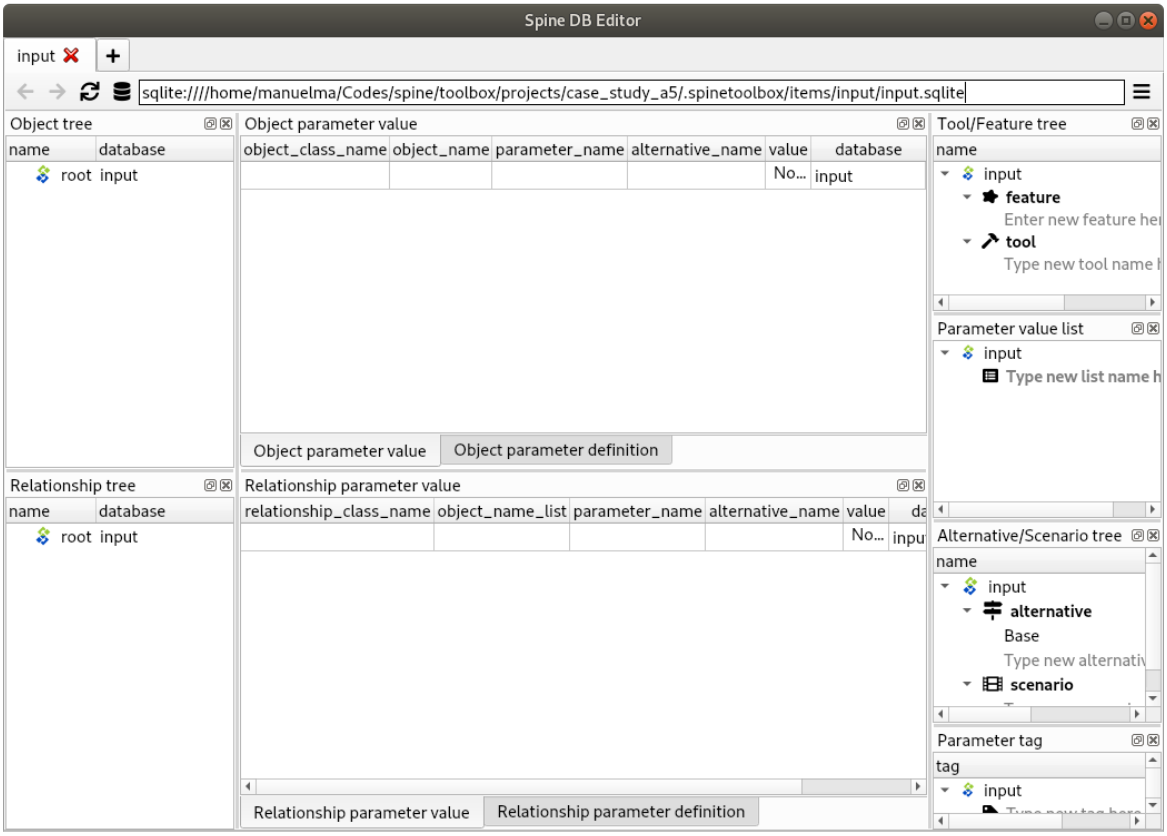


In order to run this tutorial you must first execute some preliminary steps from the Simple System tutorial. Specifically, execute all steps from the guide, up to and including the step of importing-the-spineopt-database-template. It is advisable to go through the whole tutorial in order to familiarise yourself with Spine.

Note: Just remember to give a different name for the Spine Project of the hydropower tutorial (e.g., 'Case Study A5') in the corresponding step, so to not mix up the Spine Toolbox projects!

Importing the SpineOpt database template

1. Download the SpineOpt database template (right click on the link, then select *Save link as…*)
2. Select the *input* Data Store item in the *Design View*.
3. Go to *Data Store Properties* and hit **Open editor**. This will open the newly created database in the *Spine DB Editor*, looking similar to this:



Note: The *Spine DB editor* is a dedicated interface within Spine Toolbox for visualizing and managing Spine databases.

4. Press **Alt + F** to display the editor menu, select **File -> Import…**, and then select the template file you previously downloaded (in case it is not displayed in the folder where you saved it, double-check that you selected . The contents of that file will be imported into the current database, and you should then see classes like 'commodity', 'connection' and 'model' under the root node in the *Object tree* (on the left).
5. From the editor menu (Alt + F), select **Session -> Commit**. Enter 'Import SpineOpt template' as message in the popup dialog, and click **Commit**.

Note: The SpineOpt template contains the fundamental object and relationship classes, as well as parameter definitions, that SpineOpt recognizes and expects. You can think of it as the *generic structure* of the model, as opposed to the *specific data* for a particular instance. In the remainder of this section, we will add that specific data for the Skellefte river.

3.3.2 Entering data

Note: There are two options in this tutorial to enter data in the Database. The first one is to enter data manually and the second to *use the importer* functionality. These are described in the next two subsections respectively and produce similar models. The model created when using the importer creates a model with two-segments efficiency curves for converting water to electricity (while the model created when entering the data manually assumes a simplified efficiency curve with a single segment).

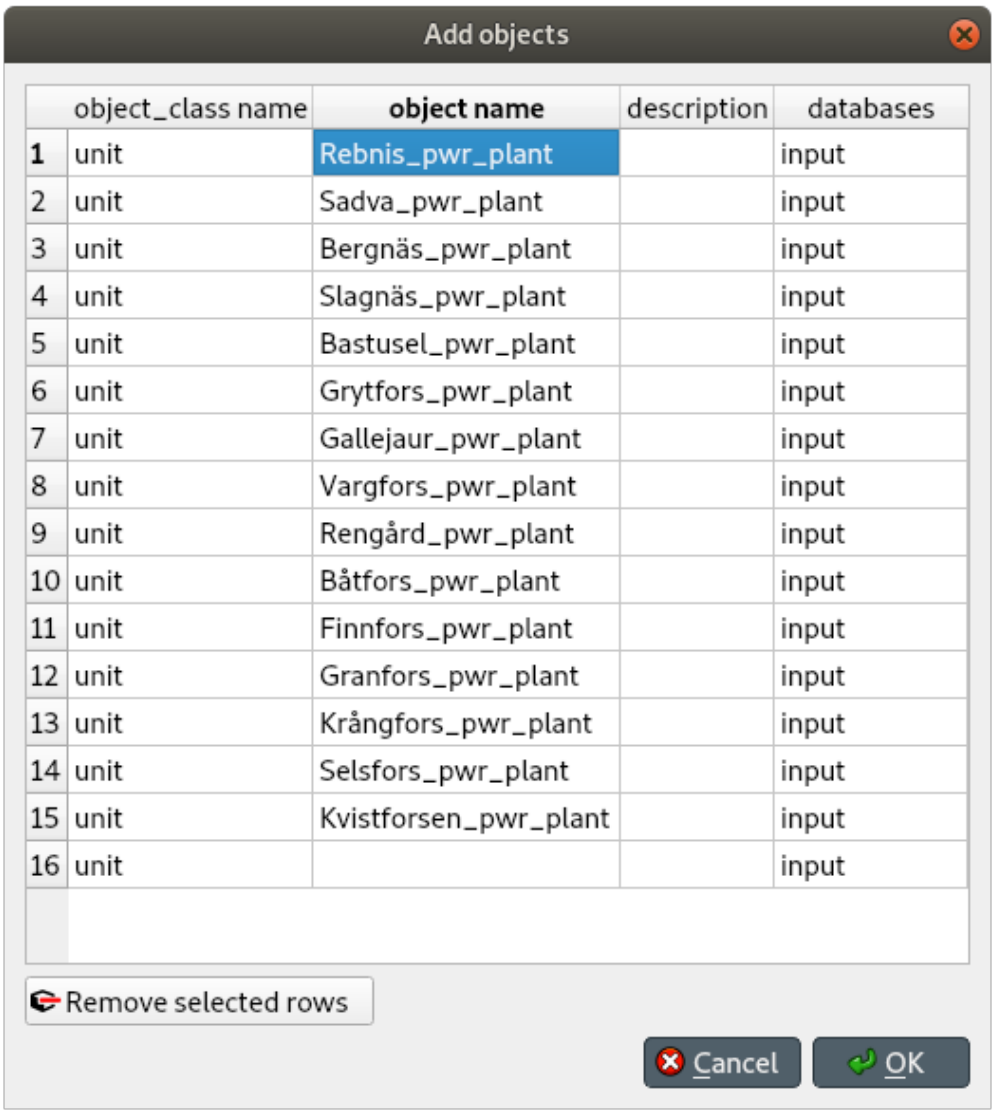
Entering data manually

Creating objects

1. To add power plants to the model, stay in the *Spine DB Editor* and create objects of class `unit` as follows:
   a. Select the list of plant names from the text-box below and copy it to the clipboard (**Ctrl+C**):

```
Rebnis_pwr_plant
Sadva_pwr_plant
Bergnäs_pwr_plant
Slagnäs_pwr_plant
Bastusel_pwr_plant
Grytfors_pwr_plant
Gallejaur_pwr_plant
Vargfors_pwr_plant
Rengård_pwr_plant
Båtfors_pwr_plant
Finnfors_pwr_plant
Granfors_pwr_plant
Krångfors_pwr_plant
Selsfors_pwr_plant
Kvistforsen_pwr_plant
```

   b. Go to *Object tree* (on the top left of the window, usually), right-click on `unit` and select **Add objects** from the context menu. This will open the *Add objects* dialog.
   c. Select the first cell under the **object name** column and press **Ctrl+V**. This will paste the list of plant names from the clipboard into that column; the **object class name** column will be filled automatically with 'unit'. The form should now be looking similar to this:



	object_class name	object name	description	databases
1	unit	Rebnis_pwr_plant		input
2	unit	Sadva_pwr_plant		input
3	unit	Bergnäs_pwr_plant		input
4	unit	Slagnäs_pwr_plant		input
5	unit	Bastusel_pwr_plant		input
6	unit	Grytfors_pwr_plant		input
7	unit	Gallejaur_pwr_plant		input
8	unit	Vargfors_pwr_plant		input
9	unit	Rengård_pwr_plant		input
10	unit	Båtfors_pwr_plant		input
11	unit	Finnfors_pwr_plant		input
12	unit	Granfors_pwr_plant		input
13	unit	Krångfors_pwr_plant		input
14	unit	Selsfors_pwr_plant		input
15	unit	Kvistforsen_pwr_plant		input
16	unit			input

d. Click **Ok**.

e. Back in the *Spine DB Editor*, under *Object tree*, double click on `unit` to confirm that the objects are effectively there.

f. Commit changes with the message 'Add power plants'.

2. Add discharge and spillway connections by creating objects of class `connection` with the following names:

```
Rebnis_to_Bergnäs_disch
Sadva_to_Bergnäs_disch
Bergnäs_to_Slagnäs_disch
Slagnäs_to_Bastusel_disch
Bastusel_to_Grytfors_disch
Grytfors_to_Gallejaur_disch
Gallejaur_to_Vargfors_disch
Vargfors_to_Rengård_disch
Rengård_to_Båtfors_disch
Båtfors_to_Finnfors_disch
```

(continues on next page)

(continued from previous page)

```
Finnfors_to_Granfors_disch
Granfors_to_Krångfors_disch
Krångfors_to_Selsfors_disch
Selsfors_to_Kvistforsen_disch
Kvistforsen_to_downstream_disch
Rebnis_to_Bergnäs_spill
Sadva_to_Bergnäs_spill
Bergnäs_to_Slagnäs_spill
Slagnäs_to_Bastusel_spill
Bastusel_to_Grytfors_spill
Grytfors_to_Gallejaur_spill
Gallejaur_to_Vargfors_spill
Vargfors_to_Rengård_spill
Rengård_to_Båtfors_spill
Båtfors_to_Finnfors_spill
Finnfors_to_Granfors_spill
Granfors_to_Krångfors_spill
Krångfors_to_Selsfors_spill
Selsfors_to_Kvistforsen_spill
Kvistforsen_to_downstream_spill
```

3. Add water nodes by creating objects of class `node` with the following names:

```
Rebnis_upper
Sadva_upper
Bergnäs_upper
Slagnäs_upper
Bastusel_upper
Grytfors_upper
Gallejaur_upper
Vargfors_upper
Rengård_upper
Båtfors_upper
Finnfors_upper
Granfors_upper
Krångfors_upper
Selsfors_upper
Kvistforsen_upper
Rebnis_lower
Sadva_lower
Bergnäs_lower
Slagnäs_lower
Bastusel_lower
Grytfors_lower
Gallejaur_lower
Vargfors_lower
Rengård_lower
Båtfors_lower
Finnfors_lower
Granfors_lower
Krångfors_lower
Selsfors_lower
```

(continues on next page)

(continued from previous page)

```
Kvistforsen_lower
```

4. Next, create the following objects (all names in **lower-case**):
   a. `instance` of class `model`.
   b. `water` and `electricity` of class `commodity`.
   c. `electricity_node` of class `node`.
   d. `electricity_load` of class `unit`.
   e. `some_week` of class `temporal_block`.
   f. `deterministic` of class `stochastic_structure`.
   g. `realization` of class `stochastic_scenario`.
5. Finally, create the following objects to get results back from Spine Opt (again, all names in **lower-case**):
   a. `my_report` of class `report`.
   b. `unit_flow`, `connection_flow`, and `node_state` of class `output`.

Note: To modify an object after you enter it, right click on it and select **Edit...** from the context menu.

Specifying object parameter values

1. To specify the general behaviour of our model, stay in the *Spine DB Editor* and enter model parameter values as follows:
   a. Select the model parameter value data from the text-box below and copy it to the clipboard (**Ctrl+C**):

```
model        instance        duration_unit        Base        "hour"
model        instance        model_end        Base        {"type": "date_time",
↪"data": "2019-01-08T00:00:00"}
model        instance        model_start        Base        {"type": "date_time
↪", "data": "2019-01-01T00:00:00"}
```

   b. Select `instance` in the *Object tree* and inspect the table in *Object parameter value* (on the top-center of the window, usually). Make sure that the columns in the table are ordered as follows (drag and drop columns if you need to change their order):

```
object_class_name | object_name | parameter_name | alternative_name | value |␣
↪database
```

   c. Select the first cell under `object_class_name` and press **Ctrl+V**. This will paste the model parameter value data from the clipboard into the table. The form should be looking like this:

Object parameter value

object_class_name	object_name	parameter_name	alternative_name	value	database
model	instance	duration_unit	Base	hour	input
model	instance	model_end	Base	2019-01-08 00:00:00	input
model	instance	model_start	Base	2019-01-01 00:00:00	input
model	instance			None	input

2. Specify the resolution of our temporal block `some_week` in the same way using the data below:

```
temporal_block	some_week	resolution	Base	{"type":
↪"duration", "data": "1h"}
```

3. Specify the behaviour of all system nodes with the data below, where:

   a. `demand` represents the local inflow (negative in most cases).

   b. `fix_node_state` represents fixed reservoir levels (at the beginning and the end).

   c. `has_state` indicates whether or not the node is a reservoir (true for all the upper nodes).

   d. `state_coeff` is the reservoir 'efficienty' (always 1, meaning that there aren't any loses).

   e. `node_state_cap` is the maximum level of the reservoirs.

   To do this in one single step, simply select `node` in the *Object tree* and paste the following values in the first empty cell:

```
node	Bastusel_upper	demand	Base	-0.2579768519
node	Bergnäs_upper	demand	Base	-22.29
node	Båtfors_upper	demand	Base	-2
node	Finnfors_upper	demand	Base	0
node	Gallejaur_upper	demand	Base	-15.356962963
node	Granfors_upper	demand	Base	0
node	Grytfors_upper	demand	Base	-3.78
node	Krångfors_upper	demand	Base	0
node	Kvistforsen_upper	demand	Base	-1.3273809524
node	Rebnis_upper	demand	Base	-3.68
node	Rengård_upper	demand	Base	-10.37
node	Sadva_upper	demand	Base	-5.43
node	Selsfors_upper	demand	Base	0
node	Slagnäs_upper	demand	Base	0
node	Vargfors_upper	demand	Base	-3.5584953704
node	Bastusel_upper	fix_node_state	Base	{"type": "time_
↪series", "data": {"2019-01-01T00:00:00": 5581.44, "2019-01-01T01:00:00": NaN,
↪"2019-01-07T23:00:00": 5417.28}}
node	Bergnäs_upper	fix_node_state	Base	{"type": "time_
↪series", "data": {"2019-01-01T00:00:00": 114543.6, "2019-01-01T01:00:00": NaN,
↪"2019-01-07T23:00:00": 105898.8}}
node	Båtfors_upper	fix_node_state	Base	{"type": "time_
↪series", "data": {"2019-01-01T00:00:00": 1117.2, "2019-01-01T01:00:00": NaN,
↪"2019-01-07T23:00:00": 891.1}}
node	Finnfors_upper	fix_node_state	Base	{"type": "time_
↪series", "data": {"2019-01-01T00:00:00": 234.0, "2019-01-01T01:00:00": NaN, "2019-
↪01-07T23:00:00": 234.0}}
node	Gallejaur_upper	fix_node_state	Base	{"type": "time_
```

(continues on next page)

(continued from previous page)

```
↪series", "data": {"2019-01-01T00:00:00": 1224.0, "2019-01-01T01:00:00": NaN,
↪"2019-01-07T23:00:00": 2808.0}}
node        Granfors_upper        fix_node_state        Base        {"type": "time_
↪series", "data": {"2019-01-01T00:00:00": 232.4, "2019-01-01T01:00:00": NaN, "2019-
↪01-07T23:00:00": 212.8}}
node        Grytfors_upper        fix_node_state        Base        {"type": "time_
↪series", "data": {"2019-01-01T00:00:00": 1060.8, "2019-01-01T01:00:00": NaN,
↪"2019-01-07T23:00:00": 1110.72}}
node        Krångfors_upper        fix_node_state        Base        {"type": "time_
↪series", "data": {"2019-01-01T00:00:00": 201.3, "2019-01-01T01:00:00": NaN, "2019-
↪01-07T23:00:00": 207.9}}
node        Kvistforsen_upper        fix_node_state        Base        {"type":
↪"time_series", "data": {"2019-01-01T00:00:00": 769.066666704, "2019-01-01T01:00:00
↪": NaN, "2019-01-07T23:00:00": 560.0}}
node        Rebnis_upper        fix_node_state        Base        {"type": "time_
↪series", "data": {"2019-01-01T00:00:00": 70243.509200184, "2019-01-01T01:00:00":␣
↪NaN, "2019-01-07T23:00:00": 59524.122689676}}
node        Rengård_upper        fix_node_state        Base        {"type": "time_
↪series", "data": {"2019-01-01T00:00:00": 1022.0, "2019-01-01T01:00:00": NaN,
↪"2019-01-07T23:00:00": 770.0}}
node        Sadva_upper        fix_node_state        Base        {"type": "time_
↪series", "data": {"2019-01-01T00:00:00": 99057.7777728, "2019-01-01T01:00:00":␣
↪NaN, "2019-01-07T23:00:00": 93831.111108}}
node        Selsfors_upper        fix_node_state        Base        {"type": "time_
↪series", "data": {"2019-01-01T00:00:00": 40.0, "2019-01-01T01:00:00": NaN, "2019-
↪01-07T23:00:00": 200.0}}
node        Slagnäs_upper        fix_node_state        Base        {"type": "time_
↪series", "data": {"2019-01-01T00:00:00": 384.0, "2019-01-01T01:00:00": NaN, "2019-
↪01-07T23:00:00": 537.6}}
node        Vargfors_upper        fix_node_state        Base        {"type": "time_
↪series", "data": {"2019-01-01T00:00:00": 3386.76, "2019-01-01T01:00:00": NaN,
↪"2019-01-07T23:00:00": 3847.68}}
node        Bastusel_upper        has_state        Base        true
node        Bergnäs_upper        has_state        Base        true
node        Båtfors_upper        has_state        Base        true
node        Finnfors_upper        has_state        Base        true
node        Gallejaur_upper        has_state        Base        true
node        Granfors_upper        has_state        Base        true
node        Grytfors_upper        has_state        Base        true
node        Krångfors_upper        has_state        Base        true
node        Kvistforsen_upper        has_state        Base        true
node        Rebnis_upper        has_state        Base        true
node        Rengård_upper        has_state        Base        true
node        Sadva_upper        has_state        Base        true
node        Selsfors_upper        has_state        Base        true
node        Slagnäs_upper        has_state        Base        true
node        Vargfors_upper        has_state        Base        true
node        Bastusel_upper        state_coeff        Base        1
node        Bergnäs_upper        state_coeff        Base        1
node        Båtfors_upper        state_coeff        Base        1
node        Finnfors_upper        state_coeff        Base        1
node        Gallejaur_upper        state_coeff        Base        1
```

(continues on next page)

(continued from previous page)

```
node        Granfors_upper        state_coeff        Base        1
node        Grytfors_upper        state_coeff        Base        1
node        Krångfors_upper        state_coeff        Base        1
node        Kvistforsen_upper        state_coeff        Base        1
node        Rebnis_upper        state_coeff        Base        1
node        Rengård_upper        state_coeff        Base        1
node        Sadva_upper        state_coeff        Base        1
node        Selsfors_upper        state_coeff        Base        1
node        Slagnäs_upper        state_coeff        Base        1
node        Vargfors_upper        state_coeff        Base        1
node        Bastusel_upper        node_state_cap        Base        8208
node        Bergnäs_upper        node_state_cap        Base        216120
node        Båtfors_upper        node_state_cap        Base        1330
node        Finnfors_upper        node_state_cap        Base        300
node        Gallejaur_upper        node_state_cap        Base        3600
node        Granfors_upper        node_state_cap        Base        280
node        Grytfors_upper        node_state_cap        Base        1248
node        Krångfors_upper        node_state_cap        Base        330
node        Kvistforsen_upper        node_state_cap        Base        1120
node        Rebnis_upper        node_state_cap        Base        205560
node        Rengård_upper        node_state_cap        Base        1400
node        Sadva_upper        node_state_cap        Base        168000
node        Selsfors_upper        node_state_cap        Base        500
node        Slagnäs_upper        node_state_cap        Base        768
node        Vargfors_upper        node_state_cap        Base        4008
```

Establishing relationships

Tip: To enter the same text on several cells, copy the text into the clipboard, then select all target cells and press **Ctrl+V**.

1. Create relationships of the class `unit__from_node` to represent that a power plant receives water from the station's upper water node, and that the electricity load takes electricity from the common electricity node. Both the power plants and the electricity load belong to the class `unit`.

   a. Select the list of unit and node names from below and copy it to the clipboard (**Ctrl+C**).

```
Rebnis_pwr_plant        Rebnis_upper
Sadva_pwr_plant        Sadva_upper
Bergnäs_pwr_plant        Bergnäs_upper
Slagnäs_pwr_plant        Slagnäs_upper
Bastusel_pwr_plant        Bastusel_upper
Grytfors_pwr_plant        Grytfors_upper
Gallejaur_pwr_plant        Gallejaur_upper
Vargfors_pwr_plant        Vargfors_upper
Rengård_pwr_plant        Rengård_upper
Båtfors_pwr_plant        Båtfors_upper
Finnfors_pwr_plant        Finnfors_upper
Granfors_pwr_plant        Granfors_upper
Krångfors_pwr_plant        Krångfors_upper
```

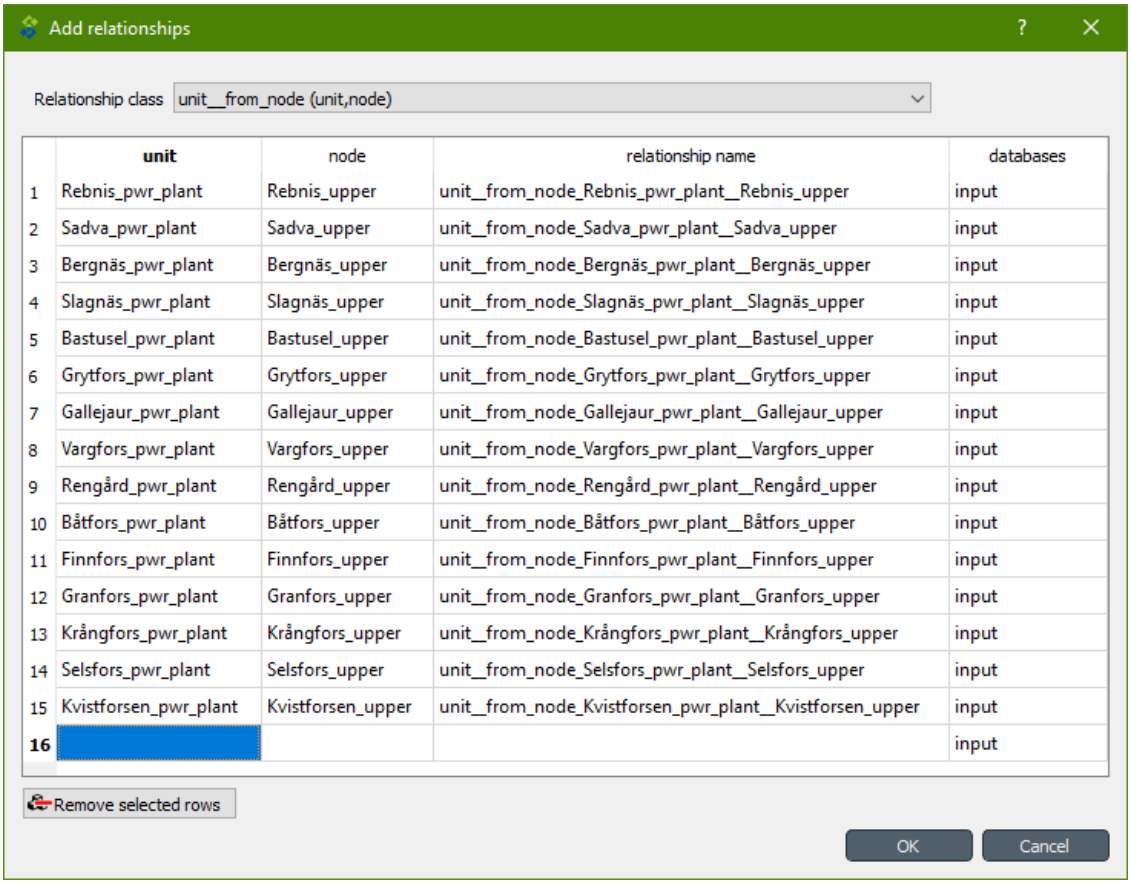
(continues on next page)

(continued from previous page)

```
Selsfors_pwr_plant        Selsfors_upper
Kvistforsen_pwr_plant        Kvistforsen_upper
electricity_load        electricity_node
```

b. In the *Spine DB Editor*, go to *Relationship tree* (on the bottom left of the window, usually), right-click on `unit__from_node` and select **Add relationships** from the context menu. This will open the *Add relationships* dialog.

c. Select the first cell under the *unit* column and press **Ctrl+V**. This will paste the list of plant and node names from the clipboard into the table. The form should be looking like this:



	unit	node	relationship name	databases
1	Rebnis_pwr_plant	Rebnis_upper	unit__from_node_Rebnis_pwr_plant__Rebnis_upper	input
2	Sadva_pwr_plant	Sadva_upper	unit__from_node_Sadva_pwr_plant__Sadva_upper	input
3	Bergnäs_pwr_plant	Bergnäs_upper	unit__from_node_Bergnäs_pwr_plant__Bergnäs_upper	input
4	Slagnäs_pwr_plant	Slagnäs_upper	unit__from_node_Slagnäs_pwr_plant__Slagnäs_upper	input
5	Bastusel_pwr_plant	Bastusel_upper	unit__from_node_Bastusel_pwr_plant__Bastusel_upper	input
6	Grytfors_pwr_plant	Grytfors_upper	unit__from_node_Grytfors_pwr_plant__Grytfors_upper	input
7	Gallejaur_pwr_plant	Gallejaur_upper	unit__from_node_Gallejaur_pwr_plant__Gallejaur_upper	input
8	Vargfors_pwr_plant	Vargfors_upper	unit__from_node_Vargfors_pwr_plant__Vargfors_upper	input
9	Rengård_pwr_plant	Rengård_upper	unit__from_node_Rengård_pwr_plant__Rengård_upper	input
10	Båtfors_pwr_plant	Båtfors_upper	unit__from_node_Båtfors_pwr_plant__Båtfors_upper	input
11	Finnfors_pwr_plant	Finnfors_upper	unit__from_node_Finnfors_pwr_plant__Finnfors_upper	input
12	Granfors_pwr_plant	Granfors_upper	unit__from_node_Granfors_pwr_plant__Granfors_upper	input
13	Krångfors_pwr_plant	Krångfors_upper	unit__from_node_Krångfors_pwr_plant__Krångfors_upper	input
14	Selsfors_pwr_plant	Selsfors_upper	unit__from_node_Selsfors_pwr_plant__Selsfors_upper	input
15	Kvistforsen_pwr_plant	Kvistforsen_upper	unit__from_node_Kvistforsen_pwr_plant__Kvistforsen_upper	input
16				input

d. Click **Ok**.

e. Back in the *Spine DB Editor*, under *Relationship tree*, double click on `unit__from_node` to confirm that the relationships are effectively there.

f. From the main menu (**Alt + F**), select **Session -> Commit** to open the *Commit changes* dialog. Enter 'Add from nodes of power plants' as the commit message and click **Commit**.

2. Create relationships of the class `unit__to_node` to represent that a power plant releases water to the station's lower water node, and that the power plants supply electricity to the common electricity node. Use the following data and do as before:

```
Rebnis_pwr_plant        Rebnis_lower
Sadva_pwr_plant        Sadva_lower
Bergnäs_pwr_plant        Bergnäs_lower
```

(continues on next page)

(continued from previous page)

```
Slagnäs_pwr_plant        Slagnäs_lower
Bastusel_pwr_plant        Bastusel_lower
Grytfors_pwr_plant        Grytfors_lower
Gallejaur_pwr_plant        Gallejaur_lower
Vargfors_pwr_plant        Vargfors_lower
Rengård_pwr_plant        Rengård_lower
Båtfors_pwr_plant        Båtfors_lower
Finnfors_pwr_plant        Finnfors_lower
Granfors_pwr_plant        Granfors_lower
Krångfors_pwr_plant        Krångfors_lower
Selsfors_pwr_plant        Selsfors_lower
Kvistforsen_pwr_plant        Kvistforsen_lower
Rebnis_pwr_plant        electricity_node
Sadva_pwr_plant        electricity_node
Bergnäs_pwr_plant        electricity_node
Slagnäs_pwr_plant        electricity_node
Bastusel_pwr_plant        electricity_node
Grytfors_pwr_plant        electricity_node
Gallejaur_pwr_plant        electricity_node
Vargfors_pwr_plant        electricity_node
Rengård_pwr_plant        electricity_node
Båtfors_pwr_plant        electricity_node
Finnfors_pwr_plant        electricity_node
Granfors_pwr_plant        electricity_node
Krångfors_pwr_plant        electricity_node
Selsfors_pwr_plant        electricity_node
Kvistforsen_pwr_plant        electricity_node
```

Note: At this point, you might be wondering what's the purpose of the `unit__node__node` relationship class. Shouldn't it be enough to have `unit__from_node` and `unit__to_node` to represent the topology of the system? The answer is yes; but in addition to topology, we also need to represent the *conversion process* that happens in the unit, where the water from one node is turned into electricty for another node. And for this purpose, we use a relationship parameter value on the `unit__node__node` relationships (see *Specifying relationship parameter values*).

3. Create relationships of the class `connection__from_node` to represent that water can be either discharged or spilled. If discharged, it is taken from the *lower* water node of the station, if spilled it is taken from the *upper* water node of the station. Use the following data and do as before:

```
Bastusel_to_Grytfors_disch        Bastusel_lower
Bergnäs_to_Slagnäs_disch        Bergnäs_lower
Båtfors_to_Finnfors_disch        Båtfors_lower
Finnfors_to_Granfors_disch        Finnfors_lower
Gallejaur_to_Vargfors_disch        Gallejaur_lower
Granfors_to_Krångfors_disch        Granfors_lower
Grytfors_to_Gallejaur_disch        Grytfors_lower
Krångfors_to_Selsfors_disch        Krångfors_lower
Kvistforsen_to_downstream_disch        Kvistforsen_lower
Rebnis_to_Bergnäs_disch        Rebnis_lower
Rengård_to_Båtfors_disch        Rengård_lower
Sadva_to_Bergnäs_disch        Sadva_lower
```

(continues on next page)

(continued from previous page)

```
Selsfors_to_Kvistforsen_disch        Selsfors_lower
Slagnäs_to_Bastusel_disch        Slagnäs_lower
Vargfors_to_Rengård_disch        Vargfors_lower
Bastusel_to_Grytfors_spill        Bastusel_upper
Bergnäs_to_Slagnäs_spill        Bergnäs_upper
Båtfors_to_Finnfors_spill        Båtfors_upper
Finnfors_to_Granfors_spill        Finnfors_upper
Gallejaur_to_Vargfors_spill        Gallejaur_upper
Granfors_to_Krångfors_spill        Granfors_upper
Grytfors_to_Gallejaur_spill        Grytfors_upper
Krångfors_to_Selsfors_spill        Krångfors_upper
Kvistforsen_to_downstream_spill        Kvistforsen_upper
Rebnis_to_Bergnäs_spill        Rebnis_upper
Rengård_to_Båtfors_spill        Rengård_upper
Sadva_to_Bergnäs_spill        Sadva_upper
Selsfors_to_Kvistforsen_spill        Selsfors_upper
Slagnäs_to_Bastusel_spill        Slagnäs_upper
Vargfors_to_Rengård_spill        Vargfors_upper
```

4. Create relationships of the class `connection__to_node` to represent that both discharge and spill are released into the *upper* node of the next downstream station. Use the following data and do as before:

```
Bastusel_to_Grytfors_disch        Grytfors_upper
Bastusel_to_Grytfors_spill        Grytfors_upper
Bergnäs_to_Slagnäs_disch        Slagnäs_upper
Bergnäs_to_Slagnäs_spill        Slagnäs_upper
Båtfors_to_Finnfors_disch        Finnfors_upper
Båtfors_to_Finnfors_spill        Finnfors_upper
Finnfors_to_Granfors_disch        Granfors_upper
Finnfors_to_Granfors_spill        Granfors_upper
Gallejaur_to_Vargfors_disch        Vargfors_upper
Gallejaur_to_Vargfors_spill        Vargfors_upper
Granfors_to_Krångfors_disch        Krångfors_upper
Granfors_to_Krångfors_spill        Krångfors_upper
Grytfors_to_Gallejaur_disch        Gallejaur_upper
Grytfors_to_Gallejaur_spill        Gallejaur_upper
Krångfors_to_Selsfors_disch        Selsfors_upper
Krångfors_to_Selsfors_spill        Selsfors_upper
Rebnis_to_Bergnäs_disch        Bergnäs_upper
Rebnis_to_Bergnäs_spill        Bergnäs_upper
Rengård_to_Båtfors_disch        Båtfors_upper
Rengård_to_Båtfors_spill        Båtfors_upper
Sadva_to_Bergnäs_disch        Bergnäs_upper
Sadva_to_Bergnäs_spill        Bergnäs_upper
Selsfors_to_Kvistforsen_disch        Kvistforsen_upper
Selsfors_to_Kvistforsen_spill        Kvistforsen_upper
Slagnäs_to_Bastusel_disch        Bastusel_upper
Slagnäs_to_Bastusel_spill        Bastusel_upper
Vargfors_to_Rengård_disch        Rengård_upper
Vargfors_to_Rengård_spill        Rengård_upper
```

Note: At this point, you might be wondering what's the purpose of the `connection__node__node` relationship

class. Shouldn't it be enough to have `connection__from_node` and `connection__to_node` to represent the topology of the system? The answer is yes; but in addition to topology, we also need to represent the *delay* in the river branches. And for this purpose, we use a relationship parameter value on the `connection__node__node` relationships (see *Specifying relationship parameter values*).

5. Create relationships of the class `node__commodity` to represent that each node has to be in balance, for water nodes with respect to water, for electricity nodes with respect to electricity. This way, you link all nodes to either the commocity `water` or the commodity `electricity`. Use the following data and do as before:

```
Rebnis_upper        water
Sadva_upper        water
Bergnäs_upper        water
Slagnäs_upper        water
Bastusel_upper        water
Grytfors_upper        water
Gallejaur_upper        water
Vargfors_upper        water
Rengård_upper        water
Båtfors_upper        water
Finnfors_upper        water
Granfors_upper        water
Krångfors_upper        water
Selsfors_upper        water
Kvistforsen_upper        water
Rebnis_lower        water
Sadva_lower        water
Bergnäs_lower        water
Slagnäs_lower        water
Bastusel_lower        water
Grytfors_lower        water
Gallejaur_lower        water
Vargfors_lower        water
Rengård_lower        water
Båtfors_lower        water
Finnfors_lower        water
Granfors_lower        water
Krångfors_lower        water
Selsfors_lower        water
Kvistforsen_lower        water
electricity_node        electricity
```

6. Define that all nodes in our model have to be balanced at each time step. To do this, you create a relationship of the class `model__default_temporal_block` between the model `instance` and the temporal_block `some_week` in the same way as before.

7. Define that our model is deterministic. To do this, you create a relationship of the class `model__default_stochastic_structure` between the model `instance` and the stochastic structure `deterministic`, as well as a relationship of class `stochastic_structure__stochastic_scenario` between the stochastid structure `deterministic` and the stochastic scenario `realization` in the same way as before.

8. In order to get the results from running Spine Opt written to the ouput database, create relationships of the class `report__output` between the report `my_report` and each of the following `output` objects: `unit_flow`, `connection_flow`, and `node_state`. In addition, you also need to create a relationship of the class `model__report` between the model `instance` and the report `my_report`.

Specifying parameter values of the relationships

1. Finally, the values of all parameters have to be entered. Specify the capacity of all hydropower plants, and their respective variable operating cost by entering `unit__from_node` parameter values as follows:

   a. Select the data from the text-box below and copy it to the clipboard (**Ctrl+C**):

```
unit__from_node        Bastusel_pwr_plant,Bastusel_upper        unit_
↪capacity        Base        127.5
unit__from_node        Bergnäs_pwr_plant,Bergnäs_upper        unit_
↪capacity        Base        120
unit__from_node        Båtfors_pwr_plant,Båtfors_upper        unit_
↪capacity        Base        210
unit__from_node        Finnfors_pwr_plant,Finnfors_upper        unit_
↪capacity        Base        176.25
unit__from_node        Gallejaur_pwr_plant,Gallejaur_upper        unit_
↪capacity        Base        228.75
unit__from_node        Granfors_pwr_plant,Granfors_upper        unit_
↪capacity        Base        180
unit__from_node        Grytfors_pwr_plant,Grytfors_upper        unit_
↪capacity        Base        123.75
unit__from_node        Krångfors_pwr_plant,Krångfors_upper        unit_
↪capacity        Base        180
unit__from_node        Kvistforsen_pwr_plant,Kvistforsen_upper        minimum_
↪operating_point        Base        0.0888888888889
unit__from_node        Kvistforsen_pwr_plant,Kvistforsen_upper        unit_
↪capacity        Base        225
unit__from_node        Rebnis_pwr_plant,Rebnis_upper        unit_
↪capacity        Base        60
unit__from_node        Rengård_pwr_plant,Rengård_upper        unit_
↪capacity        Base        165
unit__from_node        Sadva_pwr_plant,Sadva_upper        unit_
↪capacity        Base        52.5
unit__from_node        Selsfors_pwr_plant,Selsfors_upper        unit_
↪capacity        Base        225
unit__from_node        Slagnäs_pwr_plant,Slagnäs_upper        unit_
↪capacity        Base        120
unit__from_node        Vargfors_pwr_plant,Vargfors_upper        unit_
↪capacity        Base        232.5
unit__from_node        electricity_load,electricity_node        vom_
↪cost        Base        {"type": "time_series", "index": {"start": "2019-01-
↪01 00:00:00", "resolution": "1h", "ignore_year": false, "repeat": false},
↪"data": [-162.03, -156.36, -151.06, -153.52, -158.91, -164.02, -175.56, -283.
↪11, -278.76, -299.57, -285.28, -207.34, -194.95, -190.41, -185.4, -183.41, -
↪191.54, -202.9, -197.69, -195.33, -186.72, -178.87, -174.71, -168.75, -172.89,
↪ -172.13, -171.66, -173.27, -176.97, -179.63, -226.41, -271.96, -399.3, -402.
↪53, -353.28, -330.79, -294.54, -271.29, -248.71, -226.79, -240.93, -374.44, -
↪255.54, -210.47, -186.65, -178.21, -173.18, -166.44, -165.09, -162.91, -161.
↪11, -162.53, -166.04, -169.16, -174.28, -185.37, -195.79, -189.92, -187.74, -
↪181.96, -179.12, -178.36, -177.13, -177.03, -177.69, -184.8, -187.27, -179.49,
↪ -175.23, -172.67, -169.07, -165.37, -170.06, -171.48, -171.58, -174.14, -180.
↪3, -185.89, -195.37, -328.65, -365.91, -315.0, -242.68, -230.73, -225.33, -
↪225.8, -213.38, -207.32, -215.37, -243.81, -243.53, -215.56, -192.05, -187.88,
```

(continues on next page)

(continued from previous page)

```
↪ -181.15, -172.15, -174.16, -171.05, -170.77, -174.82, -179.62, -178.87, -191.
↪49, -229.64, -336.07, -242.07, -228.5, -201.85, -196.67, -192.34, -190.36, -
↪187.44, -186.68, -190.26, -191.21, -187.53, -179.34, -171.9, -166.53, -160.59,
↪ -166.08, -157.74, -145.36, -145.64, -147.42, -149.77, -153.33, -141.33, -145.
↪08, -150.52, -153.42, -159.43, -159.05, -149.49, -147.89, -150.52, -157.08, -
↪172.37, -174.06, -171.15, -160.46, -147.8, -141.61, -134.39, -144.41, -140.19,
↪ -138.59, -140.0, -139.53, -140.28, -144.03, -146.57, -149.39, -156.24, -157.
↪93, -156.43, -155.21, -149.86, -148.07, -147.13, -148.82, -162.53, -174.74, -
↪170.89, -163.94, -157.93, -150.7, -143.94]}
```

b. In the *Spine DB Editor*, go to *Relationship tree* (on the bottom left of the window, usually), and click on `unit__from_node`. Then, go to *Relationship parameter value* (on the bottom-center of the window, usually). Make sure that the columns in the table are ordered as follows (drag and drop columns if you need to change their order):

```
relationship_class_name | object_name_list | parameter_name | alternative_name␣
↪| value | database
```

c. Select the first cell under `relationship_class_name` and press **Ctrl+V**. This will paste the parameter value data from the clipboard into the table.

2. Specify the conversion ratio between discharged water and generated electricity for each hydropower station as well as a similar conversion rate (set to 1) to represent that water cannot be lost between upper and lower water nodes. Use the following data to enter the parameter values `unit__from_node`:

```
unit__node__node        Bastusel_pwr_plant,electricity_node,Bastusel_
↪upper        fix_ratio_out_in_unit_flow        Base        0.577810871184
unit__node__node        Bergnäs_pwr_plant,electricity_node,Bergnäs_upper        fix_
↪ratio_out_in_unit_flow        Base        0.0506329113924
unit__node__node        Båtfors_pwr_plant,electricity_node,Båtfors_upper        fix_
↪ratio_out_in_unit_flow        Base        0.148282097649
unit__node__node        Finnfors_pwr_plant,electricity_node,Finnfors_
↪upper        fix_ratio_out_in_unit_flow        Base        0.180985725828
unit__node__node        Gallejaur_pwr_plant,electricity_node,Gallejaur_
↪upper        fix_ratio_out_in_unit_flow        Base        0.730442000415
unit__node__node        Granfors_pwr_plant,electricity_node,Granfors_
↪upper        fix_ratio_out_in_unit_flow        Base        0.164556962025
unit__node__node        Grytfors_pwr_plant,electricity_node,Grytfors_
↪upper        fix_ratio_out_in_unit_flow        Base        0.190257000384
unit__node__node        Krångfors_pwr_plant,electricity_node,Krångfors_
↪upper        fix_ratio_out_in_unit_flow        Base        0.274261603376
unit__node__node        Kvistforsen_pwr_plant,electricity_node,Kvistforsen_
↪upper        fix_ratio_out_in_unit_flow        Base        0.472573839662
unit__node__node        Rebnis_pwr_plant,electricity_node,Rebnis_upper        fix_
↪ratio_out_in_unit_flow        Base        0.810126582278
unit__node__node        Rengård_pwr_plant,electricity_node,Rengård_upper        fix_
↪ratio_out_in_unit_flow        Base        0.165707710012
unit__node__node        Sadva_pwr_plant,electricity_node,Sadva_upper        fix_
↪ratio_out_in_unit_flow        Base        0.448462929476
unit__node__node        Selsfors_pwr_plant,electricity_node,Selsfors_
↪upper        fix_ratio_out_in_unit_flow        Base        0.209282700422
unit__node__node        Slagnäs_pwr_plant,electricity_node,Slagnäs_upper        fix_
↪ratio_out_in_unit_flow        Base        0.0443037974684
```

(continues on next page)

(continued from previous page)

```
unit__node__node        Vargfors_pwr_plant,electricity_node,Vargfors_
↪upper        fix_ratio_out_in_unit_flow        Base        0.34299714169
unit__node__node        Bastusel_pwr_plant,Bastusel_lower,Bastusel_upper        fix_
↪ratio_out_in_unit_flow        Base        1
unit__node__node        Bergnäs_pwr_plant,Bergnäs_lower,Bergnäs_upper        fix_
↪ratio_out_in_unit_flow        Base        1
unit__node__node        Båtfors_pwr_plant,Båtfors_lower,Båtfors_upper        fix_
↪ratio_out_in_unit_flow        Base        1
unit__node__node        Finnfors_pwr_plant,Finnfors_lower,Finnfors_upper        fix_
↪ratio_out_in_unit_flow        Base        1
unit__node__node        Gallejaur_pwr_plant,Gallejaur_lower,Gallejaur_
↪upper        fix_ratio_out_in_unit_flow        Base        1
unit__node__node        Granfors_pwr_plant,Granfors_lower,Granfors_upper        fix_
↪ratio_out_in_unit_flow        Base        1
unit__node__node        Grytfors_pwr_plant,Grytfors_lower,Grytfors_upper        fix_
↪ratio_out_in_unit_flow        Base        1
unit__node__node        Krångfors_pwr_plant,Krångfors_lower,Krångfors_
↪upper        fix_ratio_out_in_unit_flow        Base        1
unit__node__node        Kvistforsen_pwr_plant,Kvistforsen_lower,Kvistforsen_
↪upper        fix_ratio_out_in_unit_flow        Base        1
unit__node__node        Rebnis_pwr_plant,Rebnis_lower,Rebnis_upper        fix_ratio_
↪out_in_unit_flow        Base        1
unit__node__node        Rengård_pwr_plant,Rengård_lower,Rengård_upper        fix_
↪ratio_out_in_unit_flow        Base        1
unit__node__node        Sadva_pwr_plant,Sadva_lower,Sadva_upper        fix_ratio_
↪out_in_unit_flow        Base        1
unit__node__node        Selsfors_pwr_plant,Selsfors_lower,Selsfors_upper        fix_
↪ratio_out_in_unit_flow        Base        1
unit__node__node        Slagnäs_pwr_plant,Slagnäs_lower,Slagnäs_upper        fix_
↪ratio_out_in_unit_flow        Base        1
unit__node__node        Vargfors_pwr_plant,Vargfors_lower,Vargfors_upper        fix_
↪ratio_out_in_unit_flow        Base        1
```

3. Specify the average discharge and spillage in the first hours of the simulation. Use the following data to enter the parameter values `connection__from_node`:

```
connection__from_node        Bastusel_to_Grytfors_disch,Bastusel_lower        fix_
↪connection_flow        Base        {"type": "time_series", "index": {"start":
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [110.7, 110.7]}
connection__from_node        Bergnäs_to_Slagnäs_disch,Bergnäs_lower        fix_
↪connection_flow        Base        {"type": "time_series", "index": {"start":
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [105.0, 105.0]}
connection__from_node        Båtfors_to_Finnfors_disch,Båtfors_lower        fix_
↪connection_flow        Base        {"type": "time_series", "index": {"start":
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [150.2, 150.2]}
connection__from_node        Finnfors_to_Granfors_disch,Finnfors_lower        fix_
↪connection_flow        Base        {"type": "time_series", "index": {"start":
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [155.3, 155.3]}
connection__from_node        Gallejaur_to_Vargfors_disch,Gallejaur_lower        fix_
↪connection_flow        Base        {"type": "time_series", "index": {"start":
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [118.2, 118.2]}
connection__from_node        Granfors_to_Krångfors_disch,Granfors_lower        fix_
```

(continues on next page)

(continued from previous page)

```
↪connection_flow        Base        {"type": "time_series", "index": {"start":
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [155.6, 155.6]}
connection__from_node        Grytfors_to_Gallejaur_disch,Grytfors_lower        fix_
↪connection_flow        Base        {"type": "time_series", "index": {"start":
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [117.2, 117.2]}
connection__from_node        Krångfors_to_Selsfors_disch,Krångfors_lower        fix_
↪connection_flow        Base        {"type": "time_series", "index": {"start":
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [156.0, 156.0]}
connection__from_node        Kvistforsen_to_downstream_disch,Kvistforsen_
↪lower        fix_connection_flow        Base        {"type": "time_series", "index
↪": {"start": "2018-12-30 00:00:00", "resolution": "2D"}, "data": [158.5, 158.5]}
connection__from_node        Rebnis_to_Bergnäs_disch,Rebnis_lower        fix_
↪connection_flow        Base        {"type": "time_series", "index": {"start":
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [21.5, 21.5]}
connection__from_node        Rengård_to_Båtfors_disch,Rengård_lower        fix_
↪connection_flow        Base        {"type": "time_series", "index": {"start":
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [141.5, 141.5]}
connection__from_node        Sadva_to_Bergnäs_disch,Sadva_lower        fix_
↪connection_flow        Base        {"type": "time_series", "index": {"start":
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [34.6, 34.6]}
connection__from_node        Selsfors_to_Kvistforsen_disch,Selsfors_
↪lower        fix_connection_flow        Base        {"type": "time_series", "index
↪": {"start": "2018-12-30 00:00:00", "resolution": "2D"}, "data": [158.4, 158.4]}
connection__from_node        Slagnäs_to_Bastusel_disch,Slagnäs_lower        fix_
↪connection_flow        Base        {"type": "time_series", "index": {"start":
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [106.6, 106.6]}
connection__from_node        Vargfors_to_Rengård_disch,Vargfors_lower        fix_
↪connection_flow        Base        {"type": "time_series", "index": {"start":
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [119.6, 119.6]}
connection__from_node        Bastusel_to_Grytfors_spill,Bastusel_upper        fix_
↪connection_flow        Base        {"type": "time_series", "index": {"start":
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [0.0, 0.0]}
connection__from_node        Bergnäs_to_Slagnäs_spill,Bergnäs_upper        fix_
↪connection_flow        Base        {"type": "time_series", "index": {"start":
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [0.0, 0.0]}
connection__from_node        Båtfors_to_Finnfors_spill,Båtfors_upper        fix_
↪connection_flow        Base        {"type": "time_series", "index": {"start":
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [0.0, 0.0]}
connection__from_node        Finnfors_to_Granfors_spill,Finnfors_upper        fix_
↪connection_flow        Base        {"type": "time_series", "index": {"start":
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [0.0, 0.0]}
connection__from_node        Gallejaur_to_Vargfors_spill,Gallejaur_upper        fix_
↪connection_flow        Base        {"type": "time_series", "index": {"start":
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [0.0, 0.0]}
connection__from_node        Granfors_to_Krångfors_spill,Granfors_upper        fix_
↪connection_flow        Base        {"type": "time_series", "index": {"start":
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [0.0, 0.0]}
connection__from_node        Grytfors_to_Gallejaur_spill,Grytfors_upper        fix_
↪connection_flow        Base        {"type": "time_series", "index": {"start":
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [0.0, 0.0]}
connection__from_node        Krångfors_to_Selsfors_spill,Krångfors_upper        fix_
↪connection_flow        Base        {"type": "time_series", "index": {"start":
```

(continues on next page)

(continued from previous page)

```
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [0.0, 0.0]}
connection__from_node        Kvistforsen_to_downstream_spill,Kvistforsen_
↪upper        fix_connection_flow        Base        {"type": "time_series", "index
↪": {"start": "2018-12-30 00:00:00", "resolution": "2D"}, "data": [0.0, 0.0]}
connection__from_node        Rebnis_to_Bergnäs_spill,Rebnis_upper        fix_
↪connection_flow        Base        {"type": "time_series", "index": {"start":
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [0.0, 0.0]}
connection__from_node        Rengård_to_Båtfors_spill,Rengård_upper        fix_
↪connection_flow        Base        {"type": "time_series", "index": {"start":
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [0.0, 0.0]}
connection__from_node        Sadva_to_Bergnäs_spill,Sadva_upper        fix_
↪connection_flow        Base        {"type": "time_series", "index": {"start":
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [0.0, 0.0]}
connection__from_node        Selsfors_to_Kvistforsen_spill,Selsfors_
↪upper        fix_connection_flow        Base        {"type": "time_series", "index
↪": {"start": "2018-12-30 00:00:00", "resolution": "2D"}, "data": [0.0, 0.0]}
connection__from_node        Slagnäs_to_Bastusel_spill,Slagnäs_upper        fix_
↪connection_flow        Base        {"type": "time_series", "index": {"start":
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [0.0, 0.0]}
connection__from_node        Vargfors_to_Rengård_spill,Vargfors_upper        fix_
↪connection_flow        Base        {"type": "time_series", "index": {"start":
↪"2018-12-30 00:00:00", "resolution": "2D"}, "data": [0.0, 0.0]}
```

4. Finally, specify the delay (the time it takes for the water to run between water nodes) and the transfer ratio (being equal to 1) of different water connections. Use the following data to enter the parameter values connection__node_node:

```
connection__node__node        Bastusel_to_Grytfors_disch,Grytfors_upper,Bastusel_
↪lower        connection_flow_delay        Base        {"type": "duration", "data
↪": "1h"}
connection__node__node        Bastusel_to_Grytfors_spill,Grytfors_upper,Bastusel_
↪upper        connection_flow_delay        Base        {"type": "duration", "data
↪": "150m"}
connection__node__node        Bergnäs_to_Slagnäs_disch,Slagnäs_upper,Bergnäs_
↪lower        connection_flow_delay        Base        {"type": "duration", "data
↪": "1h"}
connection__node__node        Bergnäs_to_Slagnäs_spill,Slagnäs_upper,Bergnäs_
↪upper        connection_flow_delay        Base        {"type": "duration", "data
↪": "1h"}
connection__node__node        Båtfors_to_Finnfors_disch,Finnfors_upper,Båtfors_
↪lower        connection_flow_delay        Base        {"type": "duration", "data
↪": "3h"}
connection__node__node        Båtfors_to_Finnfors_spill,Finnfors_upper,Båtfors_
↪upper        connection_flow_delay        Base        {"type": "duration", "data
↪": "3h"}
connection__node__node        Finnfors_to_Granfors_disch,Granfors_upper,Finnfors_
↪lower        connection_flow_delay        Base        {"type": "duration", "data
↪": "3h"}
connection__node__node        Finnfors_to_Granfors_spill,Granfors_upper,Finnfors_
↪upper        connection_flow_delay        Base        {"type": "duration", "data
↪": "3h"}
connection__node__node        Gallejaur_to_Vargfors_disch,Vargfors_upper,Gallejaur_
```

(continues on next page)

(continued from previous page)

```
↪lower        connection_flow_delay        Base        {"type": "duration", "data
↪": "30m"}
connection__node__node        Gallejaur_to_Vargfors_spill,Vargfors_upper,Gallejaur_
↪upper        connection_flow_delay        Base        {"type": "duration", "data
↪": "150m"}
connection__node__node        Granfors_to_Krångfors_disch,Krångfors_upper,Granfors_
↪lower        connection_flow_delay        Base        {"type": "duration", "data
↪": "3h"}
connection__node__node        Granfors_to_Krångfors_spill,Krångfors_upper,Granfors_
↪upper        connection_flow_delay        Base        {"type": "duration", "data
↪": "3h"}
connection__node__node        Grytfors_to_Gallejaur_disch,Gallejaur_upper,Grytfors_
↪lower        connection_flow_delay        Base        {"type": "duration", "data
↪": "15m"}
connection__node__node        Grytfors_to_Gallejaur_spill,Gallejaur_upper,Grytfors_
↪upper        connection_flow_delay        Base        {"type": "duration", "data
↪": "15m"}
connection__node__node        Krångfors_to_Selsfors_disch,Selsfors_upper,Krångfors_
↪lower        connection_flow_delay        Base        {"type": "duration", "data
↪": "3h"}
connection__node__node        Krångfors_to_Selsfors_spill,Selsfors_upper,Krångfors_
↪upper        connection_flow_delay        Base        {"type": "duration", "data
↪": "3h"}
connection__node__node        Rebnis_to_Bergnäs_disch,Bergnäs_upper,Rebnis_
↪lower        connection_flow_delay        Base        {"type": "duration", "data
↪": "2D"}
connection__node__node        Rebnis_to_Bergnäs_spill,Bergnäs_upper,Rebnis_
↪upper        connection_flow_delay        Base        {"type": "duration", "data
↪": "2D"}
connection__node__node        Rengård_to_Båtfors_disch,Båtfors_upper,Rengård_
↪lower        connection_flow_delay        Base        {"type": "duration", "data
↪": "3h"}
connection__node__node        Rengård_to_Båtfors_spill,Båtfors_upper,Rengård_
↪upper        connection_flow_delay        Base        {"type": "duration", "data
↪": "3h"}
connection__node__node        Sadva_to_Bergnäs_disch,Bergnäs_upper,Sadva_
↪lower        connection_flow_delay        Base        {"type": "duration", "data
↪": "2D"}
connection__node__node        Sadva_to_Bergnäs_spill,Bergnäs_upper,Sadva_
↪upper        connection_flow_delay        Base        {"type": "duration", "data
↪": "2D"}
connection__node__node        Selsfors_to_Kvistforsen_disch,Kvistforsen_upper,
↪Selsfors_lower        connection_flow_delay        Base        {"type": "duration
↪", "data": "3h"}
connection__node__node        Selsfors_to_Kvistforsen_spill,Kvistforsen_upper,
↪Selsfors_upper        connection_flow_delay        Base        {"type": "duration
↪", "data": "3h"}
connection__node__node        Slagnäs_to_Bastusel_disch,Bastusel_upper,Slagnäs_
↪lower        connection_flow_delay        Base        {"type": "duration", "data
↪": "4h"}
connection__node__node        Slagnäs_to_Bastusel_spill,Bastusel_upper,Slagnäs_
↪upper        connection_flow_delay        Base        {"type": "duration", "data
```

(continues on next page)

(continued from previous page)

```
↪": "4h"}
connection__node__node        Vargfors_to_Rengård_disch,Rengård_upper,Vargfors_
↪lower        connection_flow_delay        Base        {"type": "duration", "data
↪": "3h"}
connection__node__node        Vargfors_to_Rengård_spill,Rengård_upper,Vargfors_
↪upper        connection_flow_delay        Base        {"type": "duration", "data
↪": "3h"}
connection__node__node        Bastusel_to_Grytfors_disch,Grytfors_upper,Bastusel_
↪lower        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Bastusel_to_Grytfors_spill,Grytfors_upper,Bastusel_
↪upper        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Bergnäs_to_Slagnäs_disch,Slagnäs_upper,Bergnäs_
↪lower        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Bergnäs_to_Slagnäs_spill,Slagnäs_upper,Bergnäs_
↪upper        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Båtfors_to_Finnfors_disch,Finnfors_upper,Båtfors_
↪lower        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Båtfors_to_Finnfors_spill,Finnfors_upper,Båtfors_
↪upper        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Finnfors_to_Granfors_disch,Granfors_upper,Finnfors_
↪lower        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Finnfors_to_Granfors_spill,Granfors_upper,Finnfors_
↪upper        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Gallejaur_to_Vargfors_disch,Vargfors_upper,Gallejaur_
↪lower        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Gallejaur_to_Vargfors_spill,Vargfors_upper,Gallejaur_
↪upper        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Granfors_to_Krångfors_disch,Krångfors_upper,Granfors_
↪lower        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Granfors_to_Krångfors_spill,Krångfors_upper,Granfors_
↪upper        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Grytfors_to_Gallejaur_disch,Gallejaur_upper,Grytfors_
↪lower        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Grytfors_to_Gallejaur_spill,Gallejaur_upper,Grytfors_
↪upper        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Krångfors_to_Selsfors_disch,Selsfors_upper,Krångfors_
↪lower        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Krångfors_to_Selsfors_spill,Selsfors_upper,Krångfors_
↪upper        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Rebnis_to_Bergnäs_disch,Bergnäs_upper,Rebnis_
↪lower        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Rebnis_to_Bergnäs_spill,Bergnäs_upper,Rebnis_
↪upper        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Rengård_to_Båtfors_disch,Båtfors_upper,Rengård_
↪lower        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Rengård_to_Båtfors_spill,Båtfors_upper,Rengård_
↪upper        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Sadva_to_Bergnäs_disch,Bergnäs_upper,Sadva_
↪lower        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Sadva_to_Bergnäs_spill,Bergnäs_upper,Sadva_
↪upper        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Selsfors_to_Kvistforsen_disch,Kvistforsen_upper,
```

(continues on next page)

(continued from previous page)

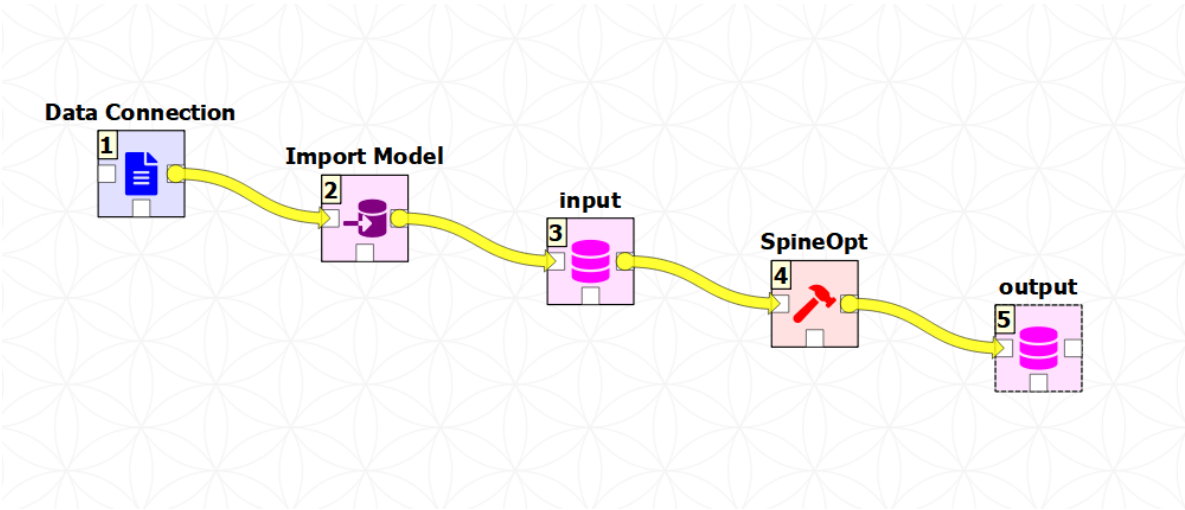
```
↪Selsfors_lower        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Selsfors_to_Kvistforsen_spill,Kvistforsen_upper,
↪Selsfors_upper        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Slagnäs_to_Bastusel_disch,Bastusel_upper,Slagnäs_
↪lower        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Slagnäs_to_Bastusel_spill,Bastusel_upper,Slagnäs_
↪upper        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Vargfors_to_Rengård_disch,Rengård_upper,Vargfors_
↪lower        fix_ratio_out_in_connection_flow        Base        1
connection__node__node        Vargfors_to_Rengård_spill,Rengård_upper,Vargfors_
↪upper        fix_ratio_out_in_connection_flow        Base        1
```

5. When you're ready, commit all changes to the database via the main menu (**Alt + F**).

Using the Importer

Additional Steps for Project Setup

1. Drag the Data Connection icon from the tool bar and drop it into the Design View. This will open the *Add Data connection dialogue*. Type in 'Data Connection' and click on **Ok**.
2. To import the model into the Spine database, you need to create an *Import specification*. Create an *Import specification* by clicking on the small arrow next to the Importer item (in the Main section of the toolbar) and press **New**. The *Importer specification editor* will pop-up.
3. Type 'Import Model' as the name of the specification. Save the specification by using **Ctrl+S** and close the window.
4. Drag the newly created Import Model Importer item icon from the tool bar and drop it into the *Design View*. This will open the Add Importer dialogue. Type in 'Import Model' and click on **Ok**.
5. Connect 'Data Connection' with 'Import Model' by first clicking on one of the Data Connection's connectors and then on one of the Importer's connectors. Connect similarly the importer with the input database. Now the project should look similar to this:



6. From the main menu, select **File -> Save project**.

Importing the model

1. Download the data and the accompanying mapping (right click on the links, then select *Save link as...*).
2. Add a reference to the file containing the model.
1. Select the *Data Connection item* in the *Design View* to show the *Data Connection properties* window (on the right side of the window usually).
2. In *Data Connection Properties*, click on , click on the icon furthest to the left **Add file references** and select the previously downloaded Excel file.
3. Next, double click on the *Import model* in the *Design view*. A window called *Select connector* for *Import Model* will pop-up, select Excel and klick **OK**. Next, still in the *Importer specification editor*, click on the main menu icon in the top right (or Press **Alt + F** to automatically display it) and import the mappings previously downloaded (by clicking on **Import mappings**). Finally, save by clicking **Ctrl+S** and exit the *Importer specification editor*.

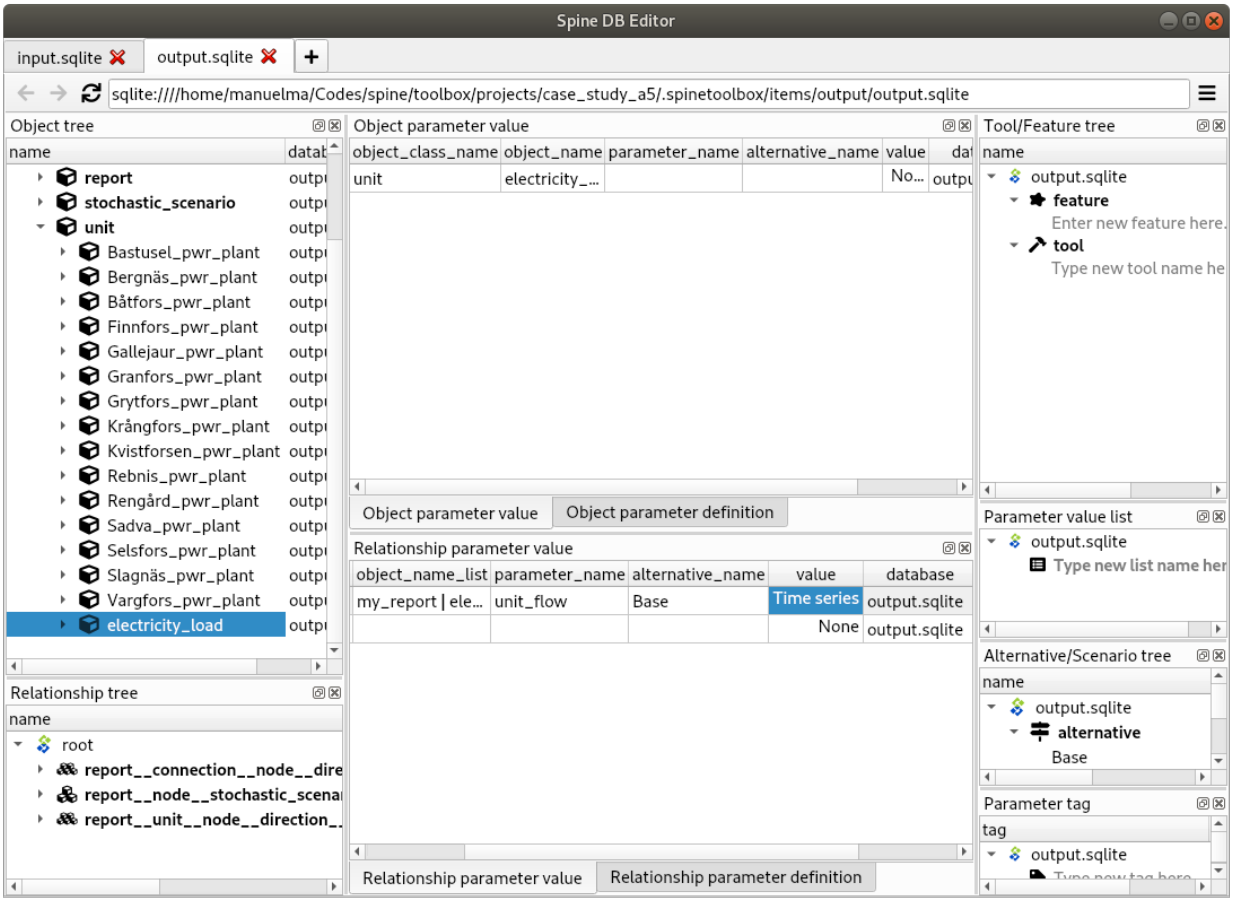
3.3.3 Executing the workflow

Once the workflow is defined and input data is in place, the project is ready to be executed. Hit the **Execute project** button on the tool bar.

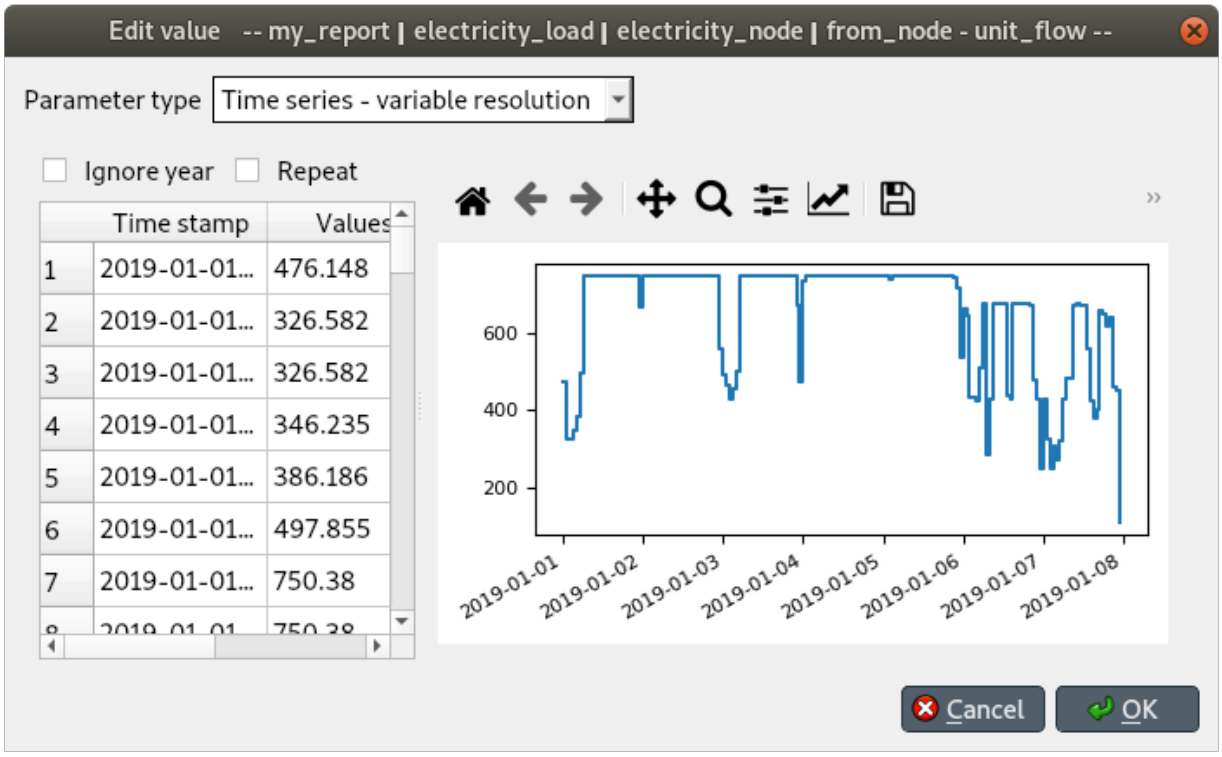
You should see 'Executing All Directed Acyclic Graphs' printed in the *Event log* (on the lower left by default). SpineOpt output messages will appear in the *Process Log* panel in the middle. After some processing, 'DAG 1/1 completed successfully' appears and the execution is complete.

3.3.4 Examining the results

Select the output data store and open the Spine DB editor.



To checkout the flow on the electricity load (i.e., the total electricity production in the system), go to *Object tree*, expand the `unit` object class, and select `electricity_load`, as illustrated in the picture above. Next, go to *Relationship parameter value* and double-click the first cell under *value*. The *Parameter value editor* will pop up. You should see something like this:



Note: If you have used the importer to instantiate the model you can easily modify the parameters in the **model** worksheet, run the project and observe the differences in the results. If you need to make changes dirrectly to the input database, in order for the importer not to overwrite them, you will need to dissassociate the importer from the input DB (right click in the connecting yellow arrow between the two items and click on **remove**).

CHAPTER

FOUR

SETTING UP EXTERNAL TOOLS

This section describes the default **Python** used by Spine Toolbox and how to change that. Here you can also find the instructions on how to set up **Julia** and **Gams** for executing Julia and Gams Tools. To get started with **SpineOpt.jl**, see *How to set up SpineOpt.jl*. See also *Executing Projects* and *Execution Modes*.

4.1 Python

No set up required! Python Tools are executed using the **default Python**, which **depends on how you installed Spine Toolbox.** The installation options are:

1. Using a single-file **installation bundle** (e.g. *spine-toolbox-0.6.0-final.2-x64.exe* or newer). You can find this file and all releases from Spine Toolbox releases. The installation bundles are only available for Windows at the moment.
2. Cloning Spine Toolbox Git repository from https://github.com/Spine-project/Spine-Toolbox. Checkout branch **release-0.6** or **master** and run *pip install -r requirements.txt* in the repo root.

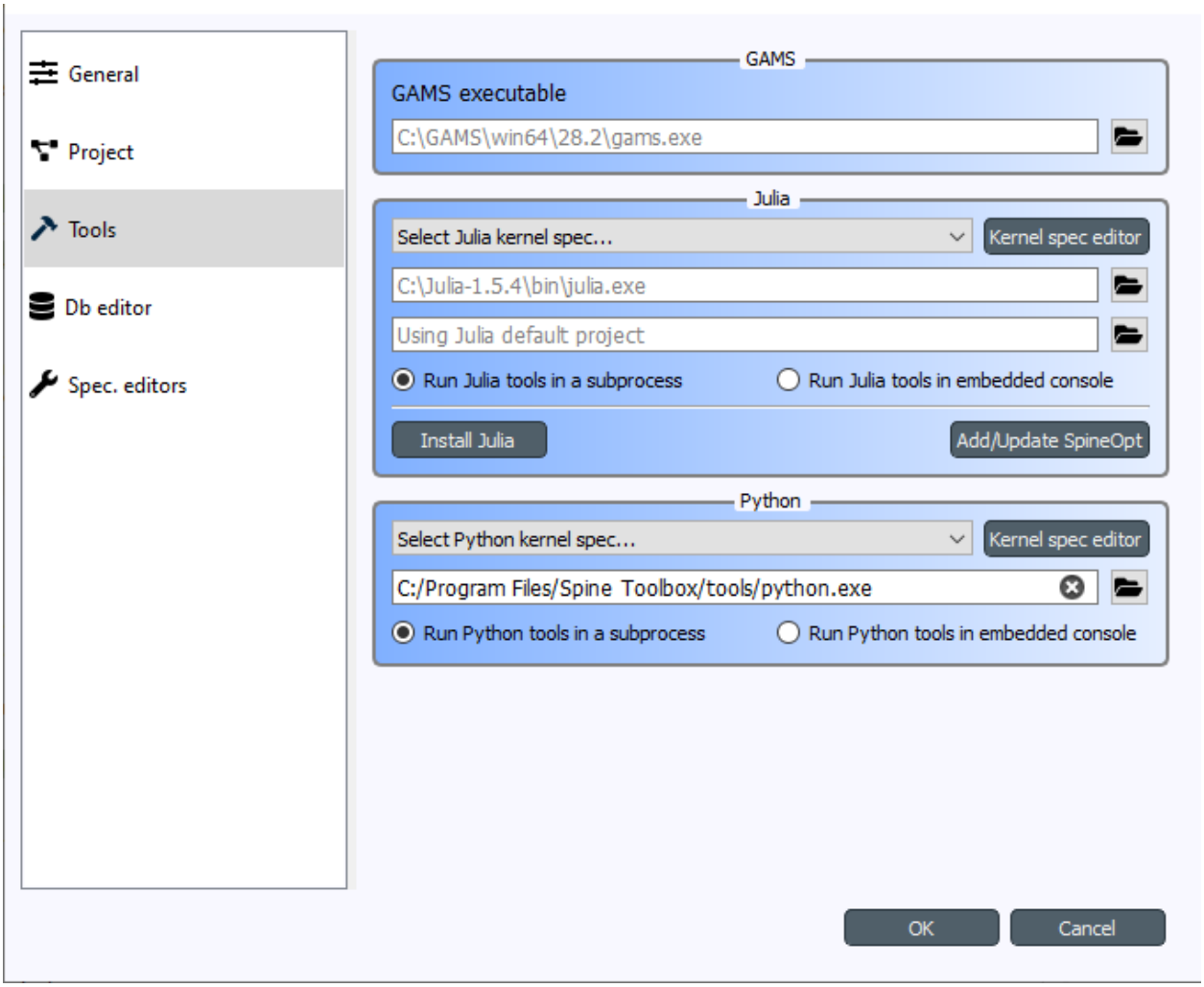
Tip: You can always see the current Python configured for Spine Toolbox from the *Tools* page in *File->Settings...*.

4.1.1 Default Python for Spine Toolbox installed using an installation bundle

The default Python is the **Python in your PATH** environment variable. **If Python is not in your PATH**, the default Python is an 'embedded' Python that is shipped with the installation bundle. The 'embedded' Python is located in *<install_dir>\tools\python.exe*, where *<install_dir>* is *C:\Program Files\Spine Toolbox* if you installed Spine Toolbox to the default directory for all users.

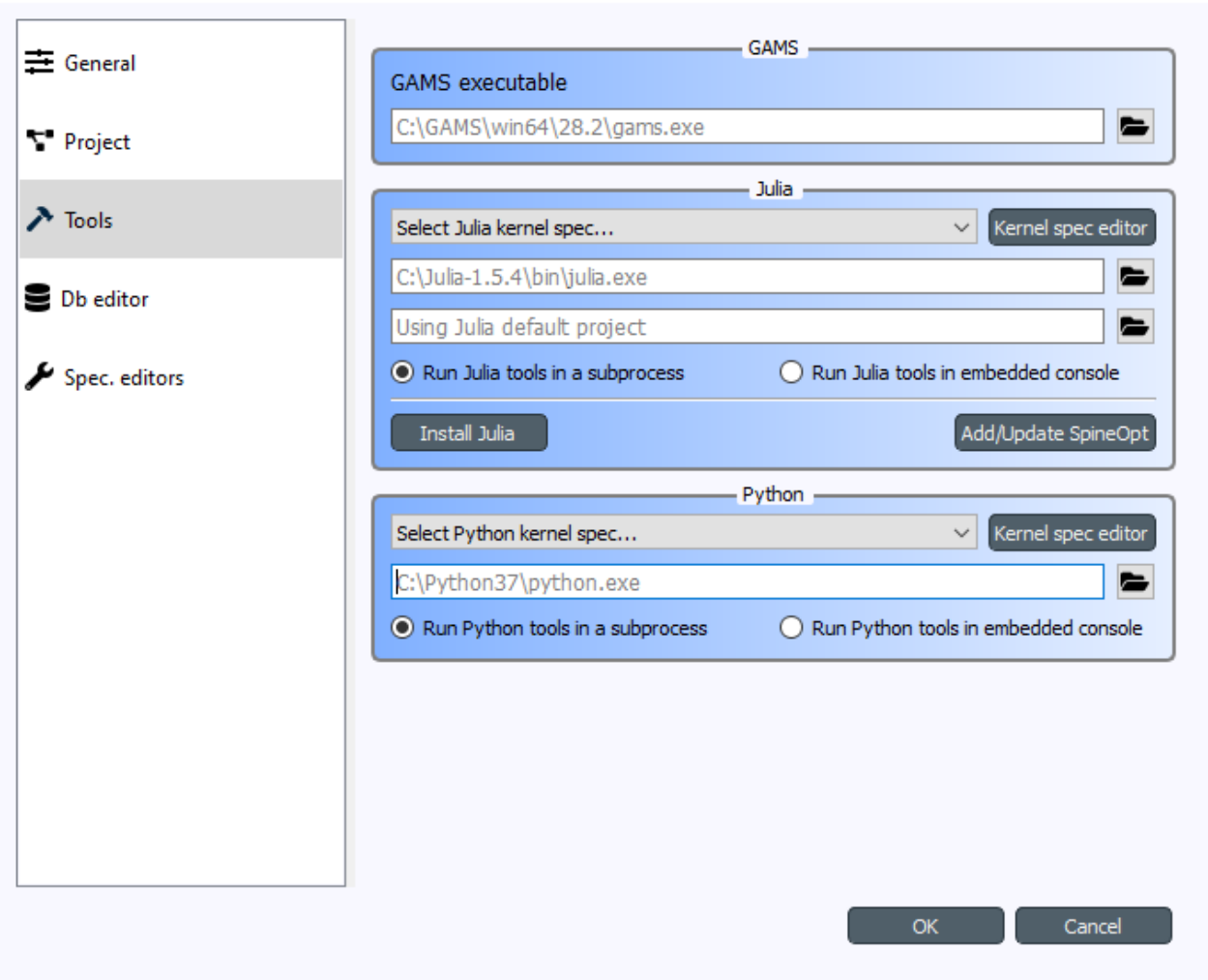
Important: If you want access to *spinedb_api* package from Tools and Consoles in Spine Toolbox, bear in mind that the version of *spinedb_api* must be compatible with the version of Spine Toolbox you are using! Spine Toolbox v0.6.0 is shipped with *spinedb_api* v0.12.1. If you want to use the Python in your PATH, **you must install the correct version of spinedb_api for this Python manually**. The correct version in this case is in the *release-0.12* branch of *spinedb_api* git repo (https://github.com/Spine-project/Spine-Database-API/tree/release-0.12). **To avoid this additional step, it is recommended** that you use the 'embedded' Python interpreter that is shipped with the application. You can set up this Python for Spine Toolbox by opening the *Tools* page of *File->Settings...* and replacing the path of the Python Interpreter with *<install_dir>\tools\python.exe*. **The 'embedded' Python interpreter has access to `spinedb_api` that is shipped with the application.**

Here are the recommended settings



4.1.2 Default Python for Spine Toolbox installed using Git

The default Python is the **Python that was used in launching the application** (i.e. *sys.executable*). When you start the app for the first time (or if you clear the path), the path to the default Python is shown as placeholder (gray) text in the line edit like this:



The default Python has access to the *spinedb_api* version that was installed with the application (the one in in <python_dir>\lib\site-packages\spinedb_api).

4.1.3 Changing the default Python

If you want to use another Python than the default, you can use existing Pythons in your system or you can download additional Pythons from https://www.python.org/downloads/. You can change the default Python on the *Tools* page of *File->Settings...* by clicking the button and selecting the Python interpreter (*python.exe* on Windows) you want. You can use **any Python in your system**.

Note: Executing Python Tools using the Jupyter Console supports Python versions from 2.7 all the way to newest one. Executing Python Tools **without** using the Jupyter Console supports even earlier Pythons than 2.7. You can start Spine Toolbox only with Python 3.7 or with 3.8, but you can set up a Jupyter Console in Spine Toolbox that uses e.g. Python 2.7. This means, that if you still have some old Python 2.7 scripts lying around, you can incorporate those into a Spine

Toolbox project workflow and execute them without modifications.

Important: If you want to have access to *spinedb_api*, you need to install it manually for the Python you select here.

4.2 Julia

Executing Julia Tools in Spine Toolbox requires that Julia is installed on your system. Julia downloads are available from https://julialang.org/downloads/. You can see the current Julia on the *Tools* page in *File->Settings...*. The **default Julia is the Julia in your PATH** environment variable. Setting some other Julia to the line edit overrides the Julia in PATH. If you want to use a specific **Julia project environment** (the place for Project.toml and Manifest.toml), you can set the path to the environment folder to the line edit just below the Julia executable line edit (the one that says *Using Julia default project* when empty).

If you are trying to execute Julia Tools and you see an error message in Event Log complaining about not finding Julia, you either don't have a Julia installation in your PATH, or the Julia path in Settings is invalid.

4.3 GAMS

Executing Gams Tools and the GDXExporter Project Item requires an installation of Gams on your system. You can download Gams from https://www.gams.com/download/.

Note: You do not need to own a Gams license as the demo version works just as well.

As with Julia, the default Gams is the Gams in your PATH environment variable. You can see the one that is currently in use from the *Tools* page in *File->Settings...*. The placeholder text shows the Gams in your PATH if found. You can also override the default Gams by setting some other gams.exe path to the line edit (e.g. *C:\GAMS\win64\28.2\gams.exe*).

Important: The bitness (32 or 64bit) of Gams has to match the bitness of the Python interpreter.

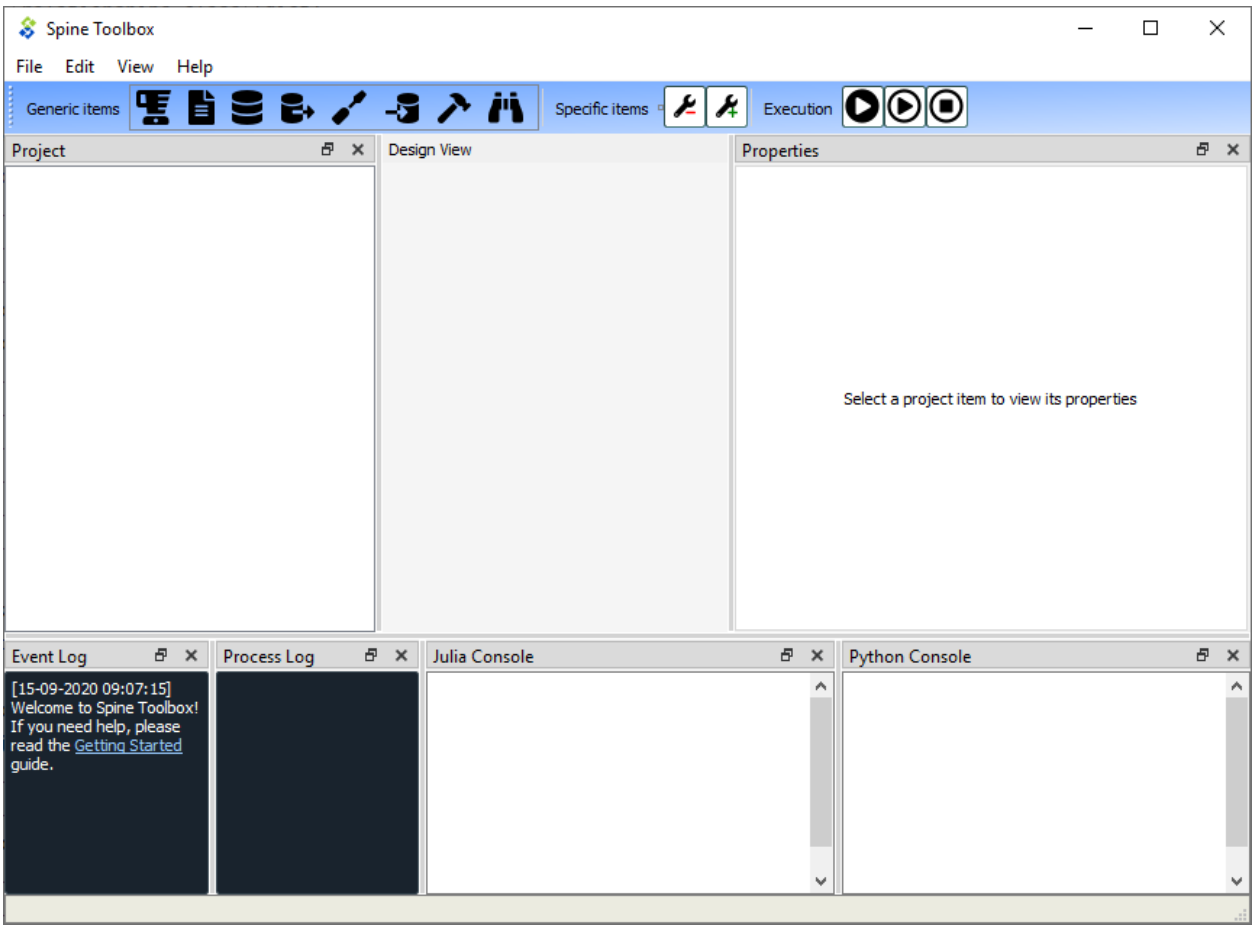
CHAPTER

FIVE

MAIN WINDOW

This section describes the different components in the application main window.

The first time you start the application you will see the main window like this.



The application main window contains six dock widgets (*Project*, *Properties*, *Event Log*, *Process Log*, *Julia Console*, and *Python Console*), a tool bar, a *Design View*, and a menu bar with *File*, *Edit*, *View*, and *Help* menus. The *Project* dock widget contains a list of project items and Tool specifications that are available in your project. The *Properties* dock widget shows the properties of the selected project item. *Event Log* shows messages depending on what you do in Spine Toolbox. *Process Log* shows messages from processes that are spawned by the application, i.e. it shows the stdout and stderr streams of GAMS, Julia, Python (if Tools are executed without embedded Julia and Python Consoles, see *Settings* section), and executable programs. Julia and Python Consoles provide full iJulia and a iPython consoles. If you choose to execute Julia tools in the embedded Julia Console, the Julia code will be included into the Julia Console

and executed there. You can interact with the iJulia in the Julia Console like you would with any iJulia you use.

Tip: You can configure the Julia and Python versions you want to use in `File->Settings`.

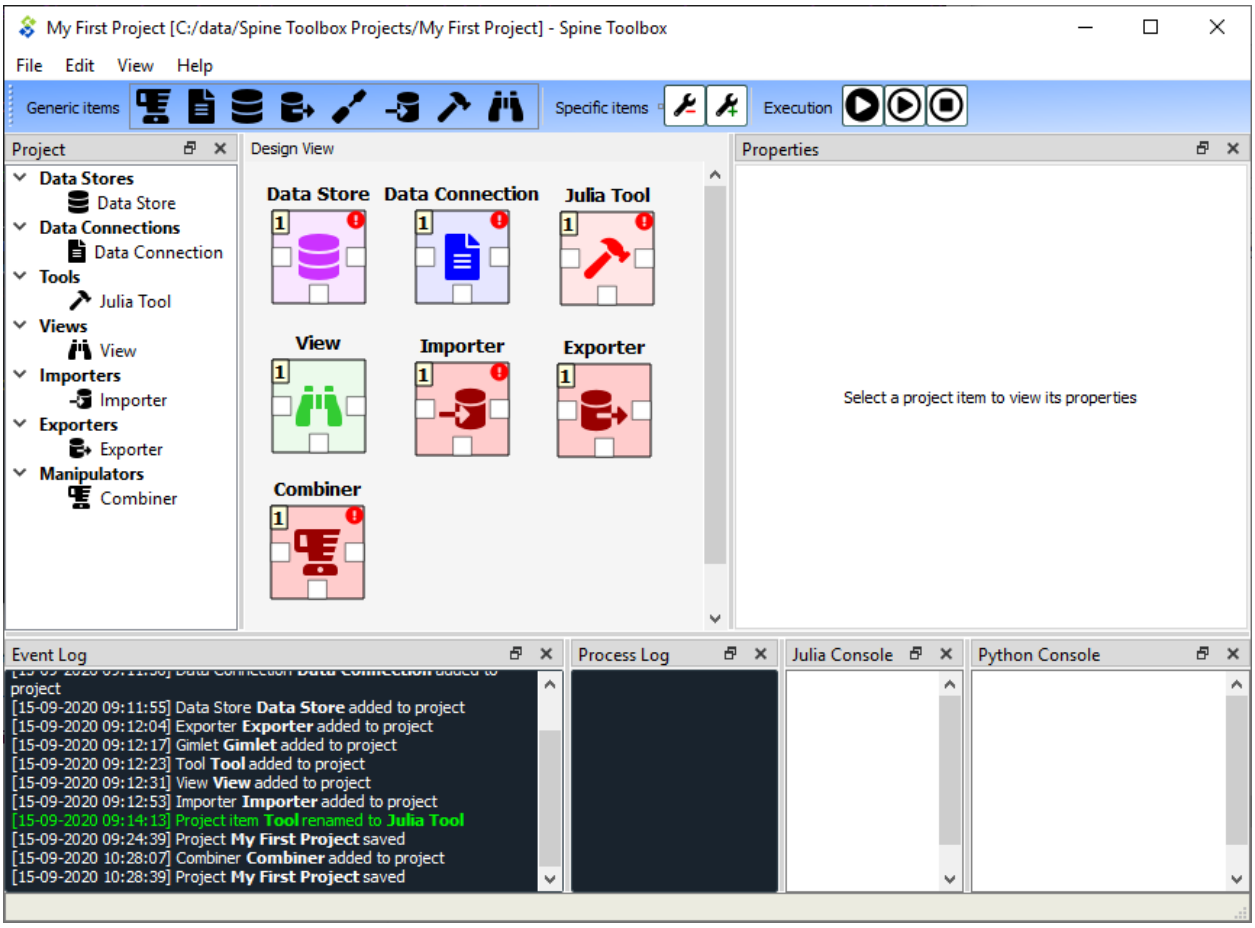
The menu bar in the top of the application contains *File*, *Edit*, *View*, and *Help* menus. In the *File* menu you can create a new project, open an existing project, save the project, upgrade an old project to modern directory-based project, and open the application Settings among other things. Spine Toolbox is project based, which means that you need to create a new project or open an existing one before you can do anything. You can create a new project by selecting `File->New project...` from the menu bar. *Drag & Drop Icon* tool bar contains the available *project item* types. The button can be used to remove all items from your project. The *Execute* icons control the execution of the items in the *Design view* where you build your project. The button executes all Directed-Acyclic Graphs (DAG) in the project in a row. The button executes the selected project items only. The button terminates the execution (if running).

You can add a new project item to your project by pointing your mouse cursor on any of the draggable items in the *Drag & Drop Icon* tool bar, then click-and-drag the item on to the *Design view*. After this you will be presented a dialog, which asks you to fill in basic information about the new project item (name, description, etc.).

The main window is very customizable so you can e.g. close the dock widgets that you do not need and/or you can resize the views to fit your needs and display size or resolution.

Note: If you want to restore all dock widgets to their default place use the menu item `View->Dock Widgets->Restore Dock Widgets`. This will show all hidden dock widgets and restore them to the main window.

Below is an example on how you can customize the main window. In the picture, a user has created a project *My First Project*, and created one project item from each of the seven categories. A Data Store called *Database*, a Data Connection called *Data files*, A Tool called *Julia model*, a View called *View*, an Importer called *Importer*, an Exporter called *Exporter*, and a Manipulator called *Combiner*. The project items are also listed in the *Project* dock widget.


My First Project [C:/data/Spine Toolbox Projects/My First Project] - Spine Toolbox
File
Edit
View
Help
Generic items
Specific items
Execution
Project
Data Stores
Data Store
Data Connections
Data Connection
Tools
Julia Tool
Views
View
Importers
Importer
Exporters
Exporter
Manipulators
Combiner
Design View
Data Store
Data Connection
Julia Tool
View
Importer
Exporter
Combiner
Properties
Select a project item to view its properties
Event Log
[15-09-2020 09:11:55] Data Store Data Store added to project
[15-09-2020 09:12:04] Exporter Exporter added to project
[15-09-2020 09:12:17] Gimlet Gimlet added to project
[15-09-2020 09:12:23] Tool Tool added to project
[15-09-2020 09:12:31] View View added to project
[15-09-2020 09:12:53] Importer Importer added to project
[15-09-2020 09:14:13] Project item Tool renamed to Julia Tool
[15-09-2020 09:24:39] Project My First Project saved
[15-09-2020 10:28:07] Combiner Combiner added to project
[15-09-2020 10:28:39] Project My First Project saved
Process Log
Julia Console
Python Console

CHAPTER

SIX

PROJECT ITEMS

Project items in the *Design view* and the connections between them make up the graph (Directed Acyclic Graph, DAG) that is executed when the or buttons are pressed.

See *Executing Projects* for more information on how a DAG is processed by Spine Toolbox. Those interested in looking under the hood can check the *Project item development* section.

6.1 Project Item Properties

Each project item has its own set of *Properties*. You can view and edit them by selecting a project item on the *Design View*. The Properties are displayed in the *Properties* dock widget on the main window. Project item properties are saved into the project save file (`project.json`), which can be found in `<proj_dir>/.spinetoolbox/` directory, where `<proj_dir>` is your current project directory.

In addition, each project item has it's own directory in the `<proj_dir>/.spinetoolbox/items/` directory. You can quickly open the project item directory in a file explorer by clicking on the button located in the lower right corner of each *Properties* form.

6.2 Project Item Descriptions

The following items are currently available:

6.2.1 Data Store

A Data store item represents a connection to a (Spine) database. Currently, the item supports sqlite and mysql dialects. The database can be accessed and modified in *Spine db editor* available by double-clicking a Data store on the Design view, from the item's properties, or from a right-click context menu.

6.2.2 Merger

A Merger item transfers data between Data Stores. When connected to a single source database, it simply copies data from the source to all output Data Stores. Data from more than one source gets merged to outputs.

6.2.3 Data Connection

A Data connection item provides access to data files. The item has two categories of files: **references** connect to files anywhere on the file system while **data** files reside in the item's own data directory.

6.2.4 Tool

Tool is the heart of a DAG. It is usually the actual model to be executed in Spine Toolbox but can be an arbitrary script, executable or system command as well. A tool is specified by its *specification*.

6.2.5 Gimlet

Note: Gimlet is pending for removal and its use in new projects is discouraged. Use Tool instead.

A Gimlet can execute an arbitrary system command with given command line arguments, input files and work directory.

6.2.6 Data Transformer

Data transformers set up database manipulators for successor items in a DAG. They do not transform data themselves; rather, Spine Database API does the transformations configured by Data transformers when the database is accessed. Currently supported transformations include entity class and parameter renaming.

6.2.7 View

A View item is meant for inspecting data from multiple sources using the *Spine db editor*. Note that the data is opened in read-only mode so modifications are not possible from the View item.

6.2.8 Importer

This item provides the user a chance to define a mapping from tabulated data such as comma separated values or Excel to the Spine data model. See *Importing and exporting data* for more information.

6.2.9 Exporter

Exporter outputs database data into tabulated file formats that can be consumed by Tool or used e.g. by external software for analysis. See *Importing and exporting data* for more information.

6.2.10 GdxExporter

Note: GdxExporter is pending for removal and its use in new projects is discouraged. Use Exporter instead.

This item exports databases contained in a *Data Store* into `.gdx` format for GAMS Tools. See *Importing and exporting data* for more information.

CHAPTER

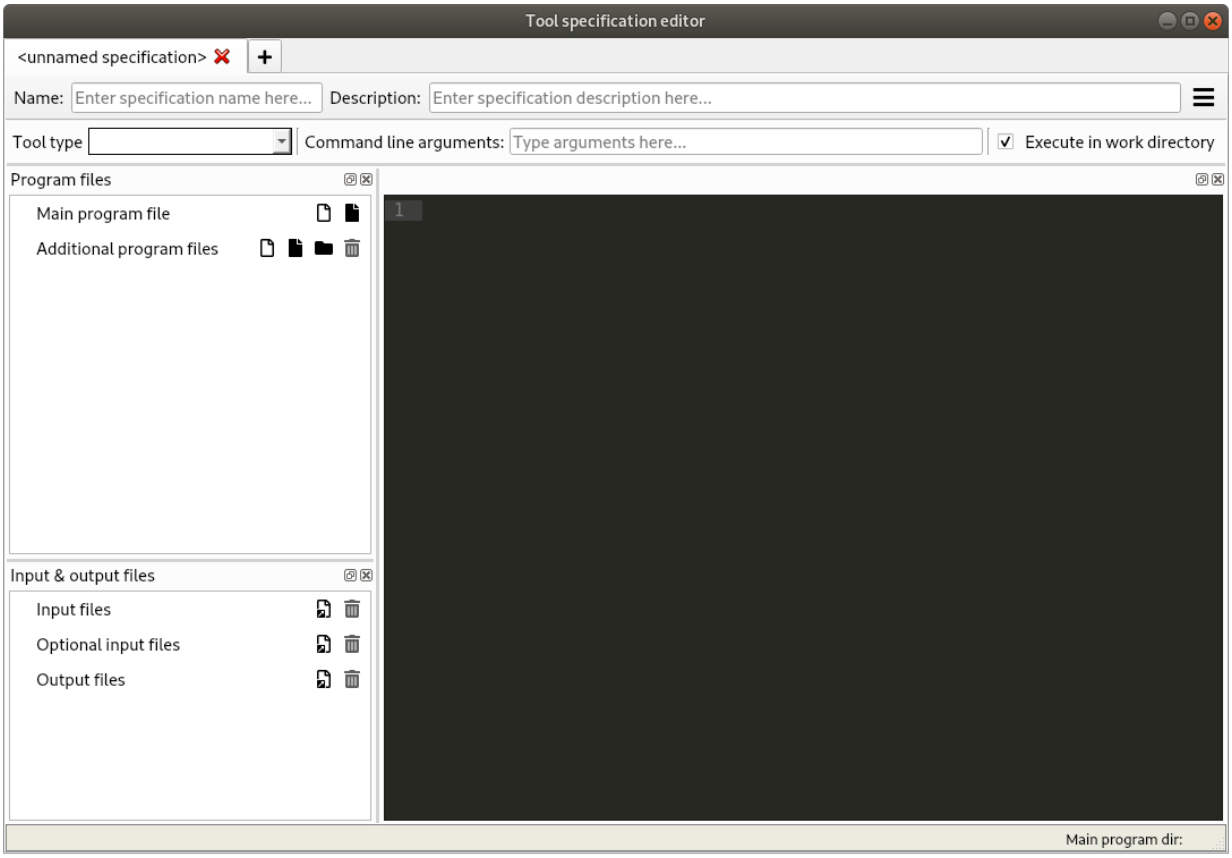
SEVEN

TOOL SPECIFICATION EDITOR

This section describes how to make a new Tool specification and how to edit existing Tool specifications.

To execute a Julia, Python, GAMS, or an executable script in Spine Toolbox, you must first create a Tool specification for your project. You can open the Tool specification editor in several ways. One way is to press the arrow next to the Tool icon in the toolbar to expand the Tool specifications, and then press the *New...* button.

When you press *New...* the following form pops up;



Start by giving the Tool specification a name. Then select the type of the Tool. You have four options (Julia, Python,

GAMS or Executable). Then select, whether you want the Tool specification to be executed in the work directory or in its source directory (See *Terminology* section). You can give the Tool specification a description, describing what the Tool specification does. Main program file is the main file of your tool, i.e. a script that can be passed to Julia, Python, GAMS, or the system shell. You can create a blank file by pressing the button, or you can browse to find an existing main program file by pressing the button.

Command line arguments can be appended to the actual command that Spine Toolbox executes in the background. For example, you may have a Windows batch file called *do_things.bat*, which accepts command line arguments *a* and *b*. Writing `a b` on the command line arguments field in the tool specification editor is the equivalent of running the batch file in command prompt with the command `do_things.bat a b`.

Additional source files is a list of files that the main program requires in order to run. You can add individual files or whole directories at once to this list.

Tip: You can also drag&drop a directory from your operating systems File Explorer into the *Additional source files* list.

Input files is a list of input data files that the program **requires** in order to execute. You can also add directories and subdirectories. Wildcards are **not** supported (see Optional input files).

Examples:

- **data.csv** -> File is copied to the same work directory as the main program
- **input/data.csv** -> Creates directory input/ to the work directory and copies file *data.csv* there
- **output/** -> Creates an empty directory output/ into the work directory

Optional input files are files that may be utilized by your program if they are found. Unix-style wildcards *?* and * are supported.

Examples:

- **data.csv** -> If found, file is copied to the same work directory as the main program
- ***.csv** -> All found .csv files are copied to the same work directory as the main program
- **input/data_?.dat** -> All found files matching the pattern *data_?.dat* are copied into input/ directory in the work directory.

Output files are files that will be archived into a timestamped result directory of the Tool's project directory after the Tool specification has finished execution. Unix-style wildcards *?* and * are supported.

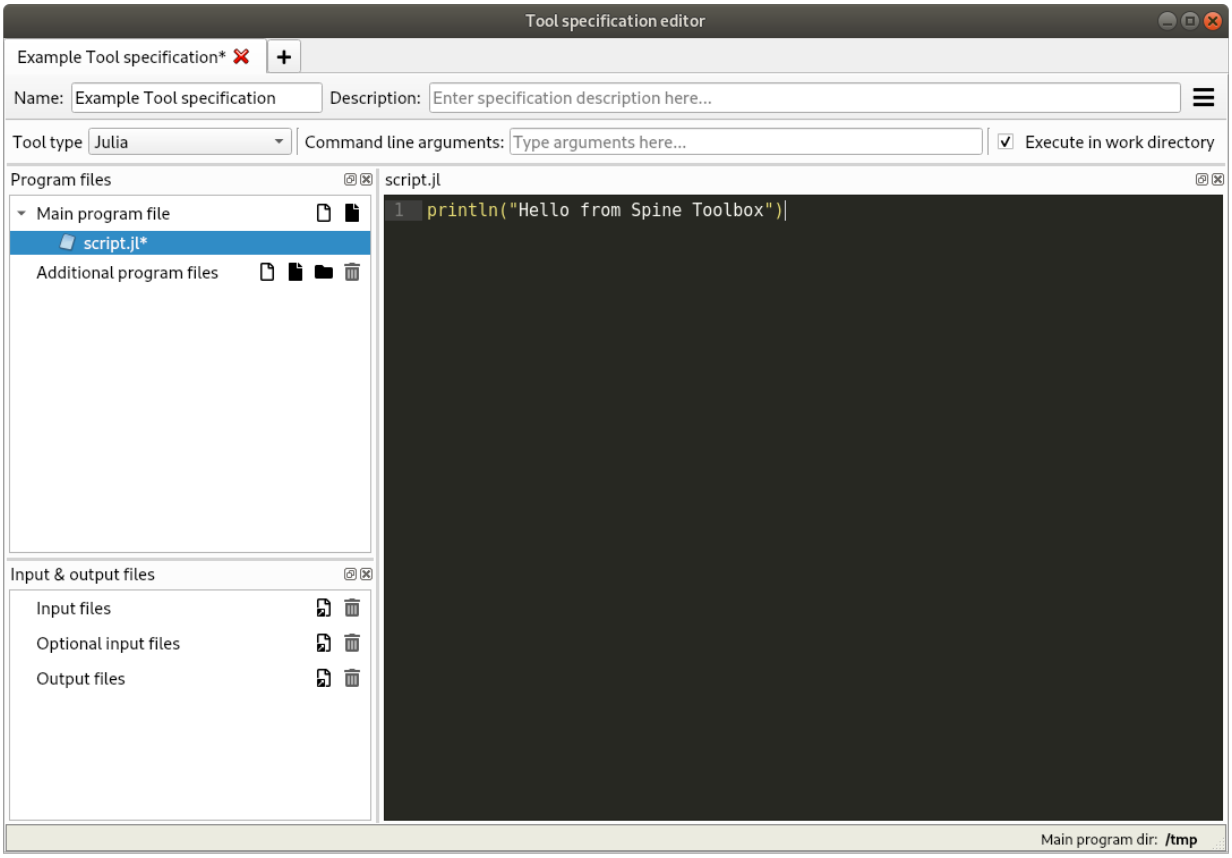
Examples:

- **results.csv** -> File is copied from work directory into results directory
- ***.csv** -> All .csv files from work directory are copied into results directory
- **output/*.gdx** -> All GDX files from the work directory's output/ subdirectory will be copied to into output/ subdirectory in the results directory.

When you are happy with your Tool specification, press **Ctrl+S** to save it. You will see a message in the Event log (back in the main Spine Toolbox window), specifying the path of the saved specification file. The Tool specification file is a text file in JSON format and has an extension *.json* You can change the location by pressing [change]. Also, you need to save your project for the specification to stick.

Tip: Only *name*, *type*, and *main program file* fields are required to make a Tool specification. The other fields are optional.

Here is a minimal Tool specification for a Julia script *script.jl*



Note: Under the hood, the contents of the Tool specification are saved to a *Tool specification file* in JSON format. Users do not need to worry about the contents of these files since reading and writing them is managed by the app. For the interested, here are the contents of the *Tool specification file* that we just created.:

```
{
    "name": "Example Tool specification",
    "description": "",
    "tooltype": "julia",
    "execute_in_work": true,
    "includes": [
        "script.jl"
    ],
    "inputfiles": [],
    "inputfiles_opt": [],
    "outputfiles": [],
    "cmdline_args": ""
}
```

After you have saved the specification, the new Tool specification has been added to the project.

To edit this Tool specification, just right-click on the Tool specification name and select *Edit specification* from the context-menu.

You are now ready to execute the Tool specification in Spine Toolbox. You just need to select a Tool item in the *Design view*, set the specification *Example Tool specification* for it, and click or button.

CHAPTER

EIGHT

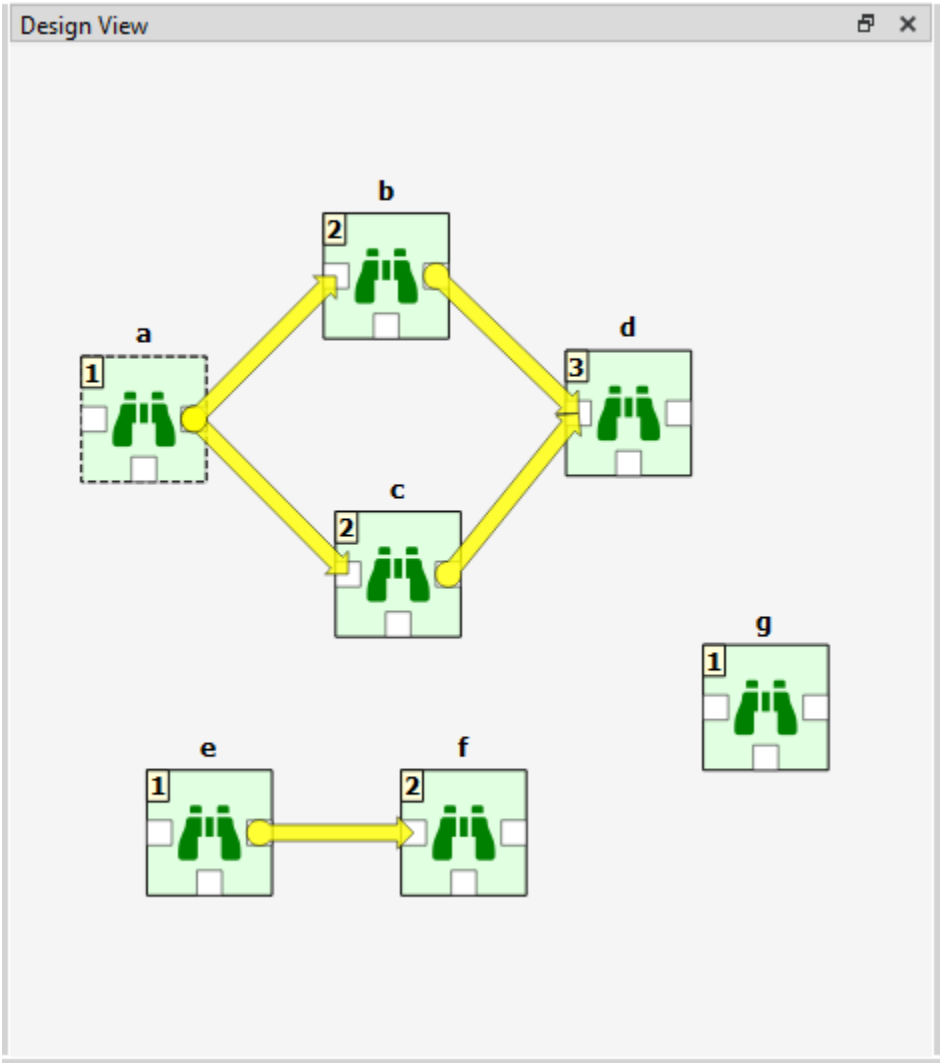
EXECUTING PROJECTS

This section describes how executing a project works and what resources are passed between project items at execution time. Execution happens by pressing the (Execute project) or the (Execute selection) buttons in the main window tool bar. A project consists of project items and connections (yellow arrows) that are visualized on the *Design View*. You use the project items and the connections to build a **Directed Acyclic Graph (DAG)**, with the project items as *nodes* and the connections as *edges*. A DAG is traversed using the **breadth-first-search** algorithm.

Rules of DAGs:

1. A single project item with no connections is a DAG.
2. All project items that are connected, are considered as a single DAG (no matter, which direction the arrows go). If there is a path between two items, they are considered as belonging to the same DAG.
3. Loops are not allowed (this is what acyclic means).

You can connect the nodes in the *Design View* how ever you want but you cannot execute the resulting DAGs if they break the rules above. Here is an example project with three DAGs.

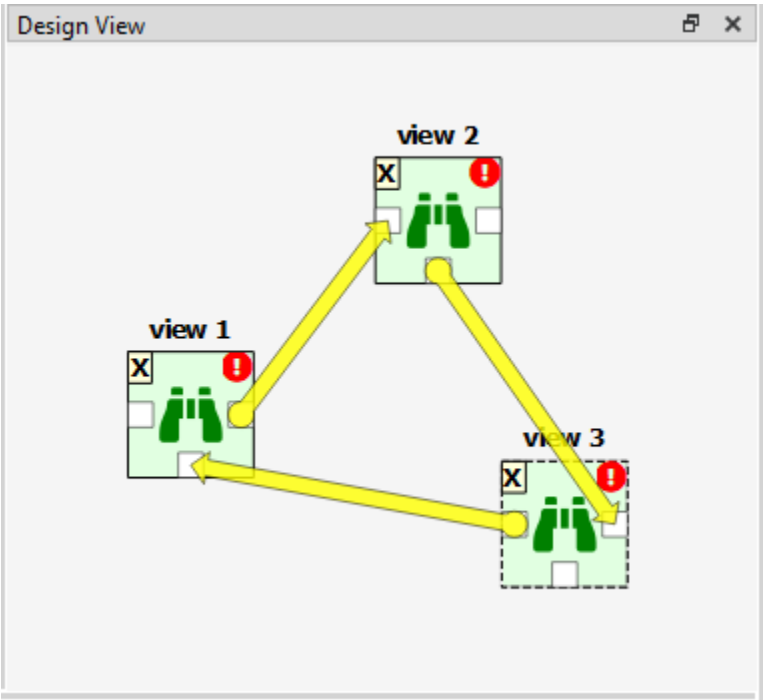


- DAG 1: items: a, b, c, d. connections: a-b, a-c, b-d, c-d
- DAG 2: items: e, f. connections: e-f
- DAG 3: items: g. connections: None

The numbers on the upper left corners of the icons show the item's **execution ranks** which roughly tell the order of execution within a DAG. Execution order of DAG 1 is *a->b->c->d* or *a->c->b->d* because b and c are **siblings** which is also indicated by their equal execution rank. DAG 2 execution order is *e->f* and DAG 3 is just *g*. All three DAGs are executed in a row though which DAG gets executed first is undefined. Therefore all DAGs have their execution ranks starting from 1.

After you press the button, you can follow the progress and the current executed item in the *Event Log*. Design view also animates the execution.

Items in a DAG that breaks the rules above are marked by X as their rank. Such DAGs are skipped during execution. The image below shows such a DAG where the items form a loop.



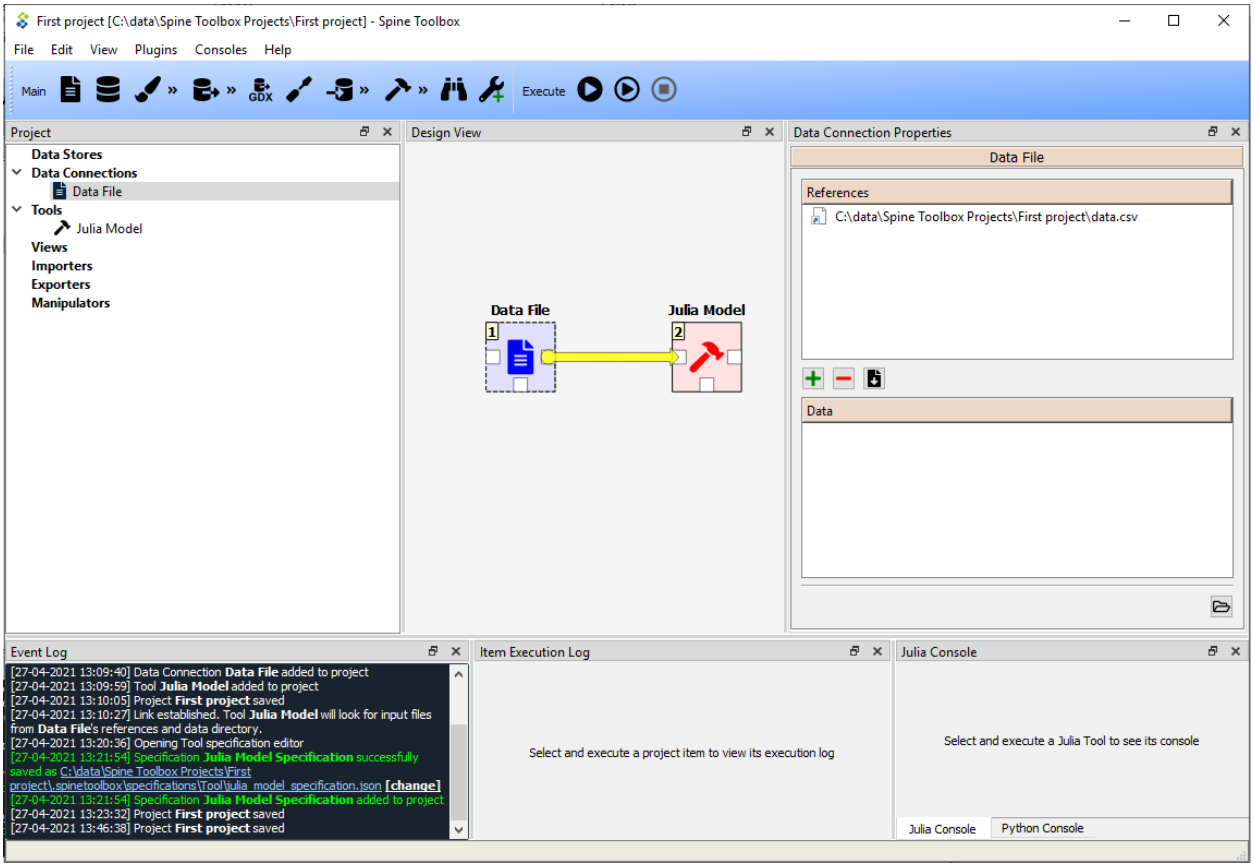
We use the words **predecessor** and **successor** to refer to project items that are upstream or downstream from a project item. **Direct predecessor** is a project item that is the immediate predecessor while **Direct Successor** is a project item that is the immediate successor. For example, in DAG 1 above, the successors of *a* are project items *b*, *c* and *d*. The direct successor of *b* is *d*. The predecessor of *b* is *a*, which is also its direct predecessor.

You can also execute only the selected parts of a project by multi-selecting the items you want to execute and pressing the button in the tool bar. For example, to execute only items *b*, *d* and *f*, select the items in *Design View* or in the project item list in *Project* dock widget and then press the button.

Tip: You can select multiple project items by holding the Ctrl-key down and clicking on desired items or by drawing a rectangle on the *Design view*.

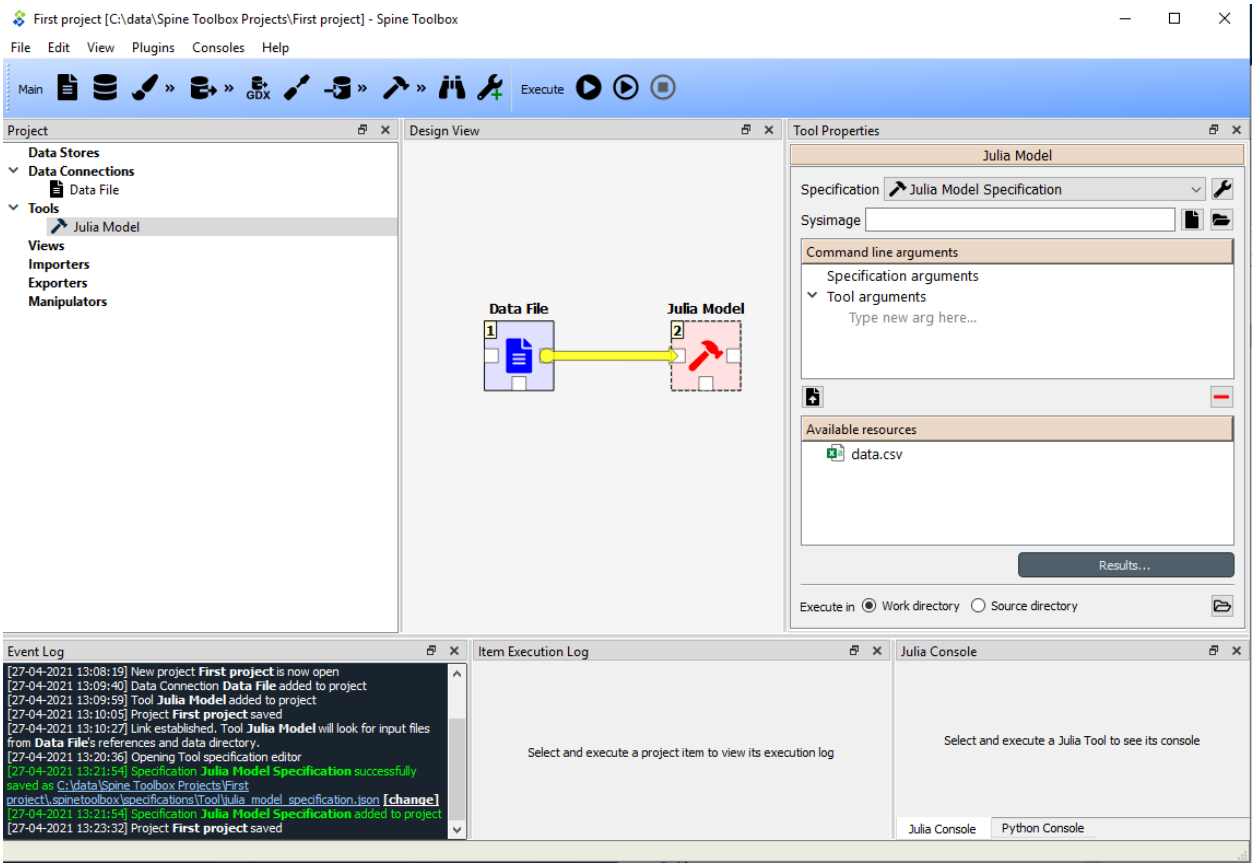
8.1 Example DAG

When you have created at least one Tool specification, you can execute a Tool as part of the DAG. The Tool specification defines the process that is executed by the Tool project item. As an example, below we have two project items; *Data File* Data Connection and *Julia Model* Tool connected to each other.



In ths example, *Data File* has a single file reference `data.csv`. Data Connections make their files visible to direct successors and thus the connection between *Data File* and *Julia Model* provides `data.csv` to the latter.

Selecting the *Julia Model* shows its properties in the *Properties* dock widget.



In the top of the Tool Properties, there is a specification drop-down menu. From this drop-down menu, you can select the Tool specification for this particular Tool item. The *Julia Model Specification* tool specification has been selected for *Julia Model*. Below the drop-down menu, you can choose a precompiled sysimage and edit Tool's command line arguments. Note that the command line argument editor already 'sees' the `data.csv` file provided by **Data File**. *Results...* button opens the Tool's result archive directory in system's file browser (all Tools have their own result directory). The *Execute in* radio buttons control, whether this Tool is first copied to a work directory and executed there, or if the execution should happen in the source directory where the main program file is located.

When you click on the button, the execution starts from the *Data File* Data Connection as indicated by the execution rank numbers. When executed, Data Connection items *advertise* their files and references to project items that are their direct successors. In this particular example, `data.csv` contained in *Data File* is also a required input file in *Julia Model Specification*. When it is the *Julia Model* tool's turn to be executed, it checks if it finds the `data.csv` from its direct predecessor items that have already been executed. Once the input file has been found the Tool starts processing the main program file *script.jl*. Note that if the connection would be the other way around (from *Julia Model* to *Data File*) execution would start from the *Julia Model* and it would fail because it cannot find the required `data.csv`. The same thing happens if there is no connection between the two project items. In this case the project items would be in separate DAGs.

Since the Tool specification type was set as *Julia* and the main program is a Julia script, Spine Toolbox starts the execution in the Julia Console (if you have selected this in the application *Settings*, See *Settings* section).

CHAPTER

NINE

EXECUTION MODES

You can execute Python or Julia Tools in the Jupyter Console or as in the shell. Gams Tools are only executed as in the shell.

9.1 Python

9.1.1 Shell execution (default)

On *Tools* page in *File->Settings...*, check the *Run Python Tools in a subprocess* radiobutton (release-0.6) or uncheck the *Jupyter Console* check box (master). This is the default execution mode for Python Tools.

9.1.2 Jupyter Console / Python Console execution

If you want to use the embedded Python Console (Jupyter Console). Check the *Run Python Tools in embedded console* radiobutton (release-0.6) or check the *Jupyter Console* check box (master). There is an extra step involved since the Jupyter Console requires a couple of extra packages (*ipykernel* and its dependencies) to be installed on the selected Python. In addition, kernel specifications for the selected Python need to be installed beforehand. **Spine Toolbox can install these for you**, from the **Kernel Spec Editor** widget that you can open from the *Tools* page in *File->Settings..* by clicking the *Kernel Spec Editor* button. In the Kernel Spec Editor, give the spec a name and click *Make kernel specification* button.

Note: You can install Python kernel specifications manually and Spine Toolbox will find them. You can select the kernel spec used in the Jupyter Console from the drop-down menu *Select Python kernel spec....*

9.2 Julia

9.2.1 Shell execution (default)

On *Tools* page in *File->Settings...*, check the *Run Julia Tools in a subprocess* radiobutton (release-0.6) or uncheck the *Jupyter Console* check box (master). This is the default execution mode for Julia Tools.

9.2.2 Jupyter Console / Julia Console execution

Like the Python Console, Julia Console requires some extra setting up. The Julia Console requires a couple of additional packages (*IJulia*, etc.) to be installed and built. **Spine Toolbox can set this up for you automatically**. Just click the **Kernel spec Editor** button, give the spec a name and click *Make kernel specification* button.

Note: You can install Julia kernel specifications manually and Spine Toolbox will find them. You can select the kernel spec used in the Jupyter Console from the drop-down menu *Select Julia kernel spec...*.

CHAPTER

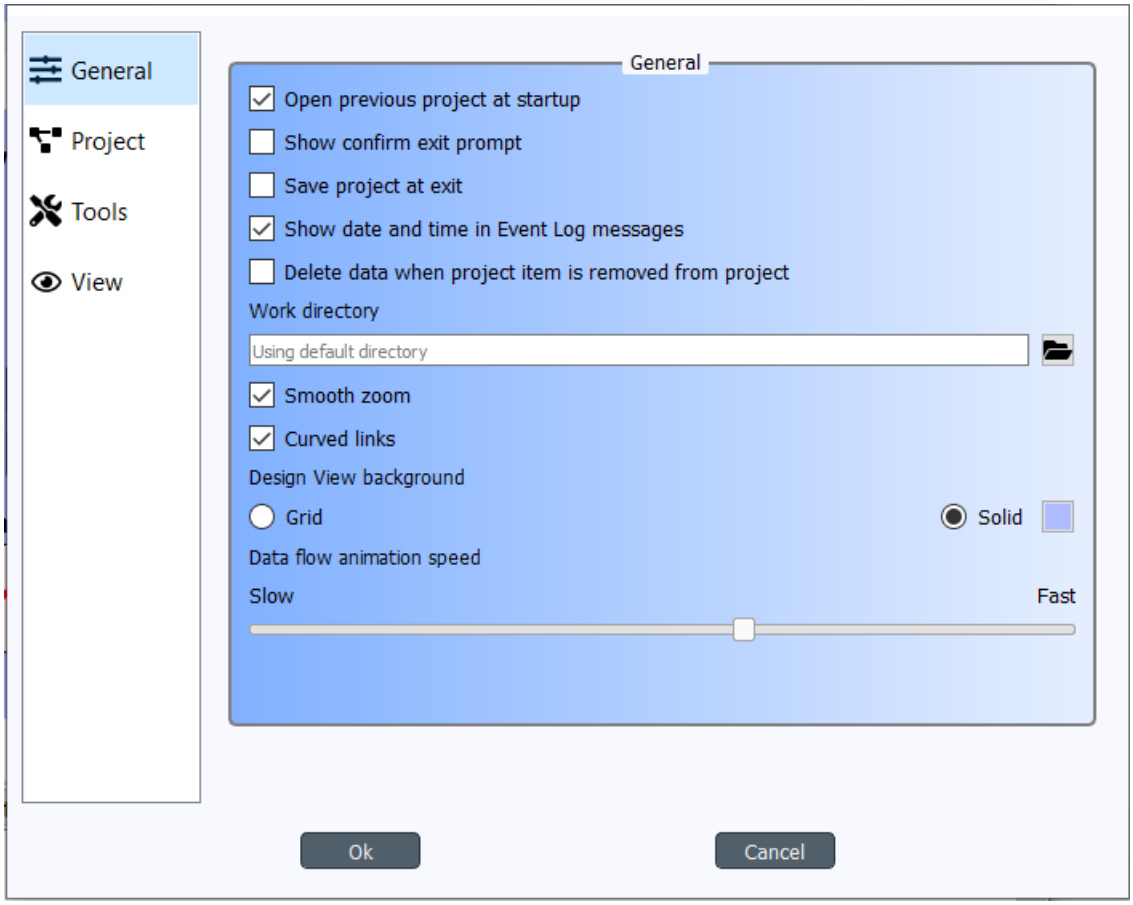
TEN

SETTINGS

Note: The images and some of the text are outdated. See tooltips in the app for up-to-date information.

You can open Spine Toolbox settings from the main window menu *File->Settings...*, or by pressing **F1**. Settings are categorized into four tabs; *General*, *Project*, *Tools*, and *View*. In addition to application settings, each Project item has user adjustable properties (See *Project Items*)

10.1 General settings

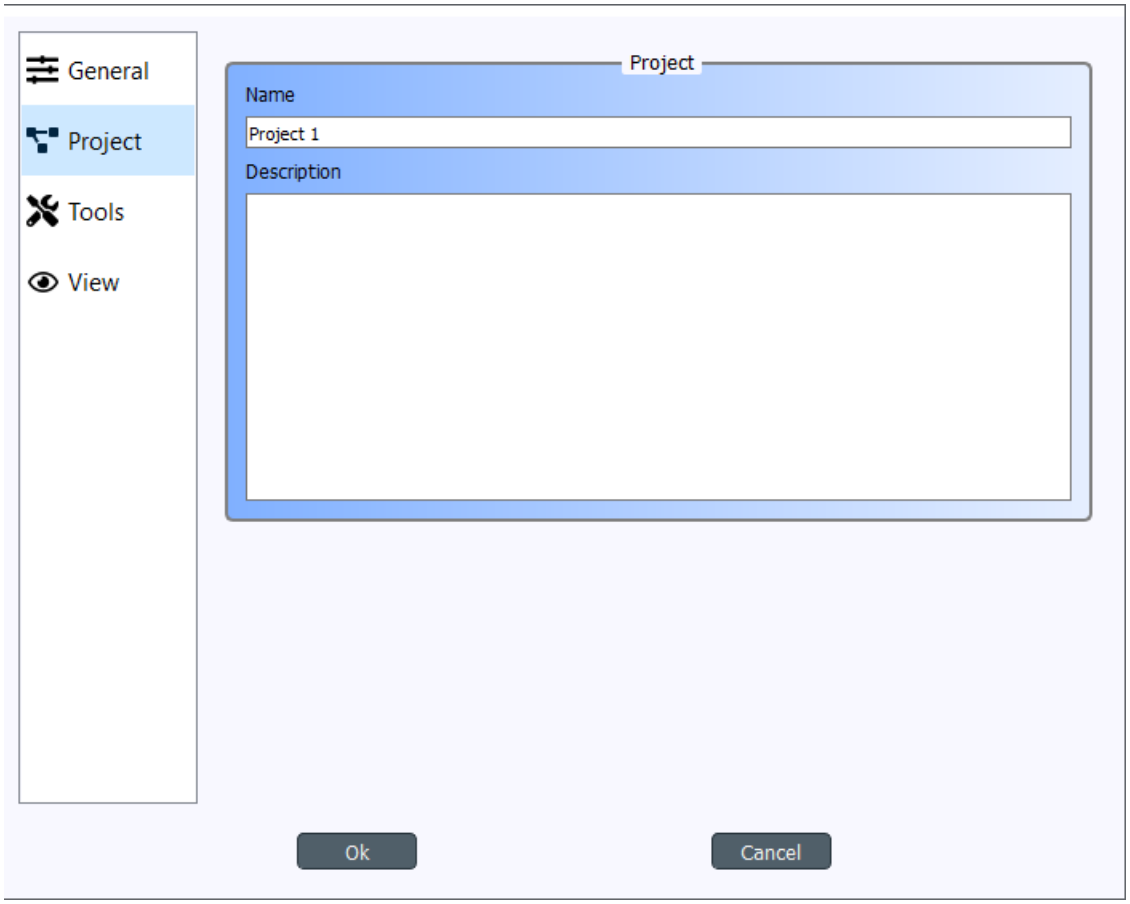


The General tab contains the general application settings.

- **Open previous project at startup** If checked, application opens the project at startup that was open the last time the application was shut down. If left unchecked, application starts without a project open.
- **Show confirm exit prompt** If checked, confirm exit prompt is shown. If unchecked, application exits without prompt.
- **Save project at exit** Unchecked: Does not save project and does not show message box. Partially checked: Shows message box (default). Checked: Saves project and does not show message box.
- **Show date and time in Event Log messages** If checked, every Event Log message is prepended with a date and time 'tag'.
- **Delete data when project item is removed from project** Check this box to delete project item's data when a project item is removed from project. This means, that the *project item directory* and its contents will be deleted from your hard drive. You can find the project item directories from the `<proj_dir>/.spinetoolbox/items/` directory, where `<proj_dir>` is your current project directory.
- **Work directory** Directory where processing the Tool takes place. Default place (if left empty) is the `/work` subdirectory of Spine Toolbox install directory. You can change this directory. Make sure to clean up the directory every now and then.
- **Smooth zoom** Controls the way zooming (by using the mouse wheel) behaves in Design View and in Spine database editor. Controls if the zoom in/out is continuous or discrete. On older computers, smooth zoom is not recommended (because it may be slower).

- **Curved links** Controls the look of the arrows (connections) on Design View.
- **Design View background** Choosing grid shows a black grid as the Design View background. Choosing Solid and then clicking on the square next to it let's you choose the background color.
- **Data flow animation speed** This slider controls the speed of the 'arrow' animation on Design View when execution is about to start.

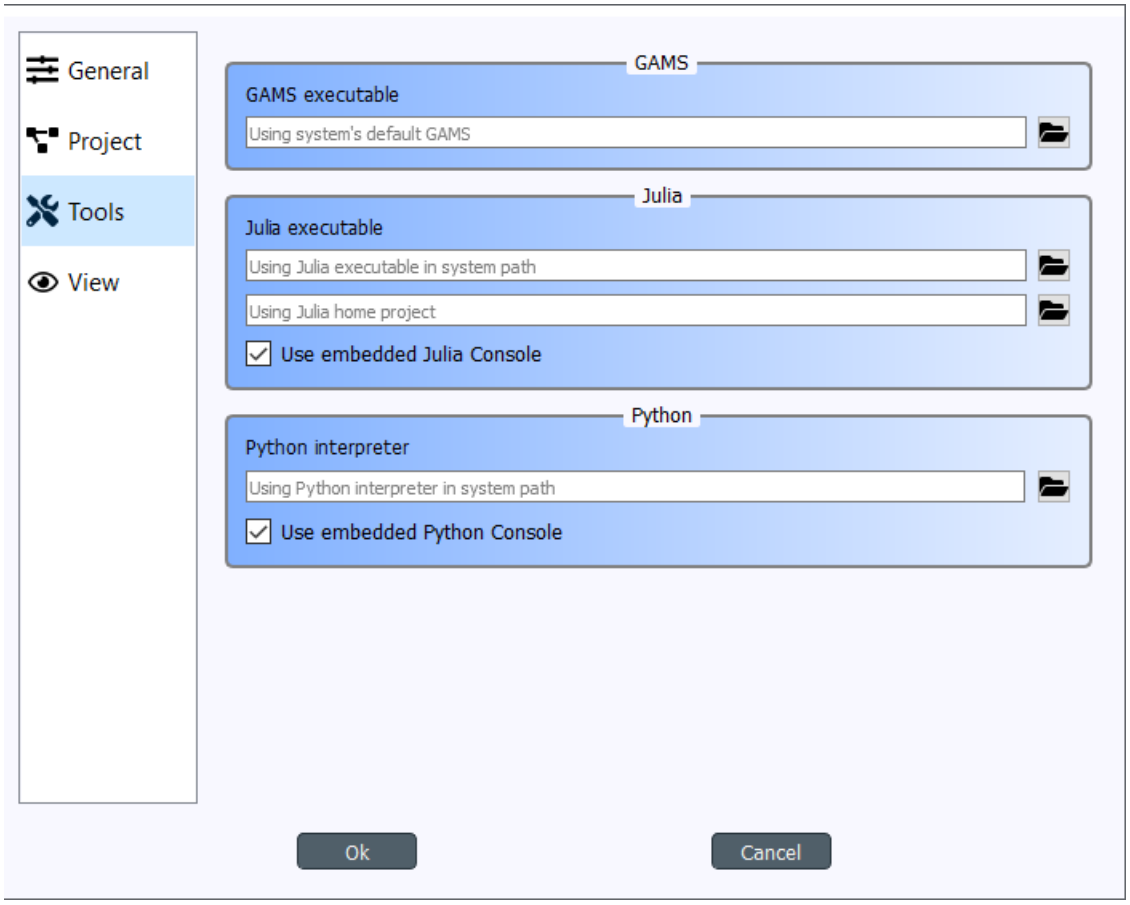
10.2 Project settings



These settings affect the project that is currently open. To save the project to a new directory use the `File->Save project as...` menu item. Or you can simply copy the project directory anywhere on your file system.

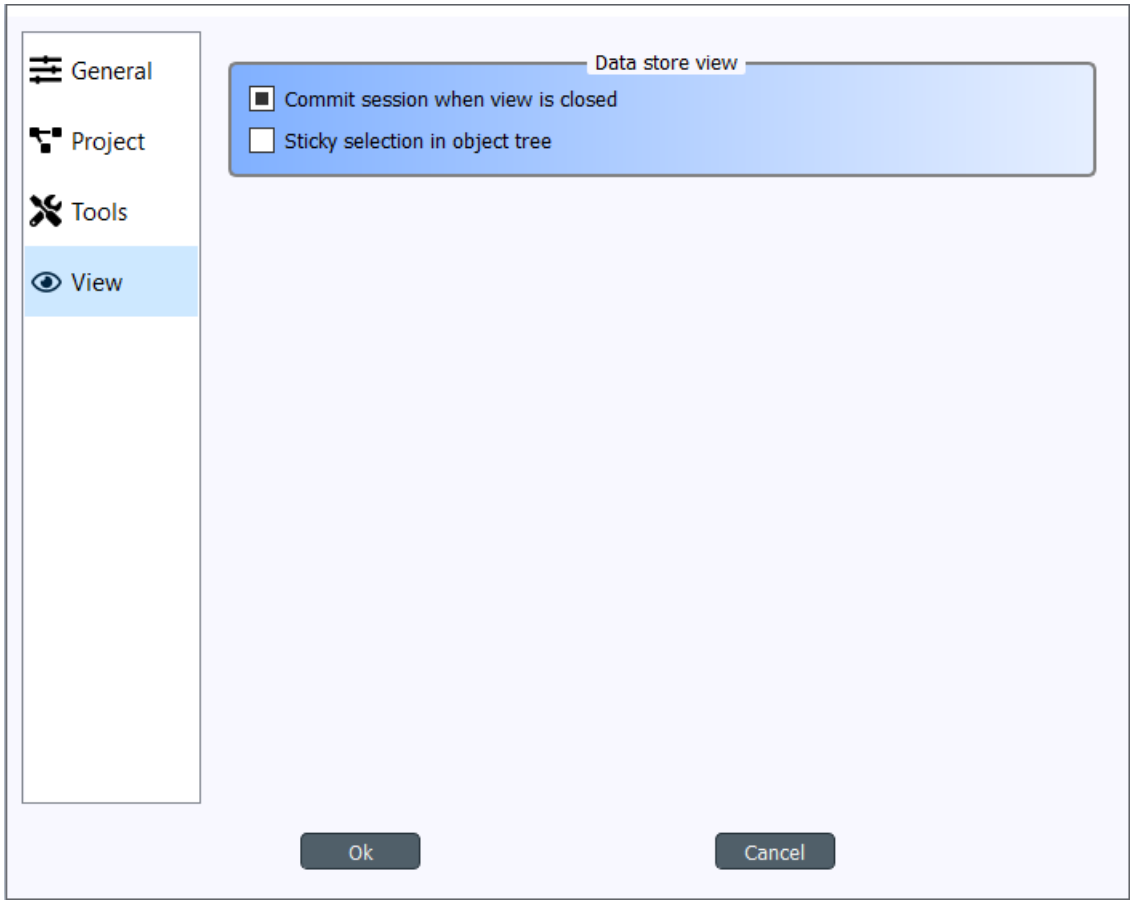
- **Name** The default name for new projects is the name of the project directory. You can change the project name here.
- **Description** You can type a description for your project here.

10.3 Tools settings



- **GAMS executable** Path to GAMS executable you wish to use to execute *GdxExporter* project items and *Tool* project items that use a GAMS Tool specification. See *Setting up External Tools*.
- **Julia executable** Path to Julia executable you wish to use to execute *Tool* project items that use a Julia Tool specification. See *Setting up External Tools*.
- **Julia home project** Set the Julia home project here.
- **Use embedded Julia Console** Check this box to execute *Tool* project items that use a Julia Tool specification in the built-in Julia Console. If you leave this un-checked, Julia Tool specifications will be executed as in the shell. I.e on Windows this would be the equivalent to running command `julia.exe example_script.jl` in the command prompt. If you decide to use the embedded Julia Console, the `example_script.jl` is *included* into the console and executed there.
- **Python interpreter** Path to Python executable you wish to use to execute *Tool* project items that use a Python Tool specification. See *Setting up External Tools*.
- **Use embedded Python Console** Check this box to execute Python Tool specifications in the embedded Python Console. If you un-check this box, Python Tool specifications will be executed as in the shell. I.e on Windows this would be the equivalent to running command `python.exe script.py` in the command prompt. If you decide to use the embedded Python Console, `script.py` is executed there instead.

10.4 View settings



- **Commit session when view is closed** This checkbox controls what happens when you close the Spine database editor which has uncommitted changes. When this is unchecked, all changes are discarded without notice. When this is partially checked (default), a message box warning you about uncommitted changes is shown. When this is checked, a commit message box is shown immediately without first showing the message box.
- **Sticky selection in object tree** Controls how selecting items in Spine database editor's Object tree using the left mouse button works. If unchecked, single selection is enabled and pressing the Ctrl-button down enables multiple selection. If checked, Multiple selection is enabled and pressing the Ctrl-button down enables single selection.

10.5 Application preferences

Spine Toolbox remembers the size, location, and placement of most of the application windows from the previous session (i.e. when closing and restarting the app).

10.6 Where are the application settings stored?

Application settings and preferences (see above) are saved to a location that depends on your operating system. On Windows, they are stored into registry key `HKEY_CURRENT_USER\Software\SpineProject\Spine Toolbox`. It is safe to delete this key if you want to reset Spine Toolbox to factory defaults.

Note: If you are looking for information on project item properties, see *Project Items*.

CHAPTER

ELEVEN

WELCOME TO SPINE DATABASE EDITOR'S USER GUIDE!

Spine database editor is a dedicated component of Spine Toolbox, that you can use to visualize and edit data in one or more Spine databases.

11.1 Spine data structure

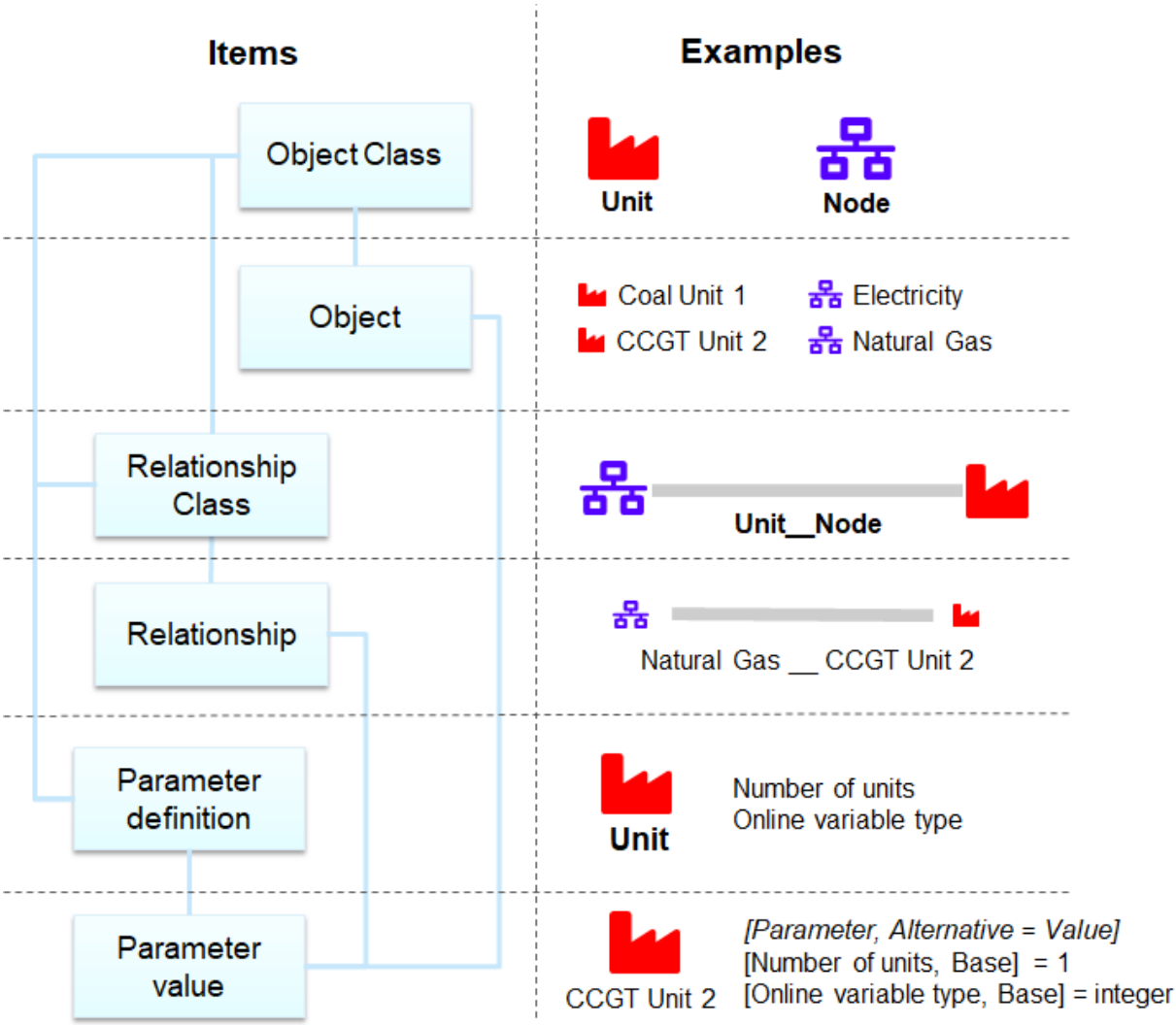
11.1.1 Main features

Spine data structure follows entity-attribute-value (EAV) with classes and relationships data model (Wikipedia). It is an open schema where the data structure is defined through data (and not through database structure). Spine Toolbox also adds an ability to hold alternative parameter values for the same parameter of a particular entity. This allows the creation of scenarios. A potential weakness of EAV is that each parameter value needs a separate row in the database which could make the parameter table large and slow. In Spine Toolbox this is circumvented by allowing different datatypes like time series and maps to be represented in the parameter field and thus greatly reducing the number of rows required to present large systems.

Definitions

1. Entity: an object (one dimension) or a relationship (n-dimensions)
2. Attribute: parameter name
3. Value: parameter value
4. Entity class: a category for entities (e.g. 'unit' is an object class while 'coal_power_plant' is an entity of 'unit' class)
5. Alternative: Each parameter value belongs to one alternative
6. Scenario: Combines alternatives into a single scenario

Diagram



11.2 Getting started

- *Launching the editor*
  - *From Spine Toolbox*
  - *From the command line*
- *Knowing the UI*

11.2.1 Launching the editor

From Spine Toolbox

To open a single database in Spine database editor:

1. Create a *Data Store* project item.
2. Select the *Data Store*.
3. Enter the url of the database in *Data Store Properties*.
4. Press the **Open editor** button in *Data Store Properties*.

To open multiple databases in Spine database editor:

1. Repeat steps 1 to 3 above for each database.
2. Create a *View* project item.
3. Connect each *Data Store* item to the *View* item.
4. Select the *View* item.
5. Press **Open editor** in *View Properties*.

From the command line

To open a single database in Spine database editor, use the `spine-db-editor` application which comes with Spine Toolbox:

```
spine-db-editor "...url of the database..."
```

Note that for e.g. an SQLite database, the url should start with 'sqlite:'.

11.2.2 Knowing the UI

The form has the following main UI components:

- *Entity trees* (*Object tree* and *Relationship tree*): they present the structure of classes and entities in all databases in the shape of a tree.
- *Stacked tables* (*Object parameter value*, *Object parameter definition*, *Relationship parameter value*, and *Relationship parameter definition*): they present object and relationship parameter data in the form of stacked tables.
- *Pivot table* and *Frozen table*: they present data for a given class in the form of a pivot table, optionally with frozen dimensions.
- *Entity graph*: it presents the structure of classes and entities in the shape of a graph.
- *Tool/Feature tree*: it presents tools, features, and methods defined in the databases.
- *Parameter value list*: it presents parameter value lists available in the databases.
- *Alternative/Scenario tree*: it presents scenarios and alternatives defined in the databases.
- *Parameter tag*: it presents parameter tags defined in the databases.

Tip: You can customize the UI from the **View** and **Pivot** sections in the hamburger menu.

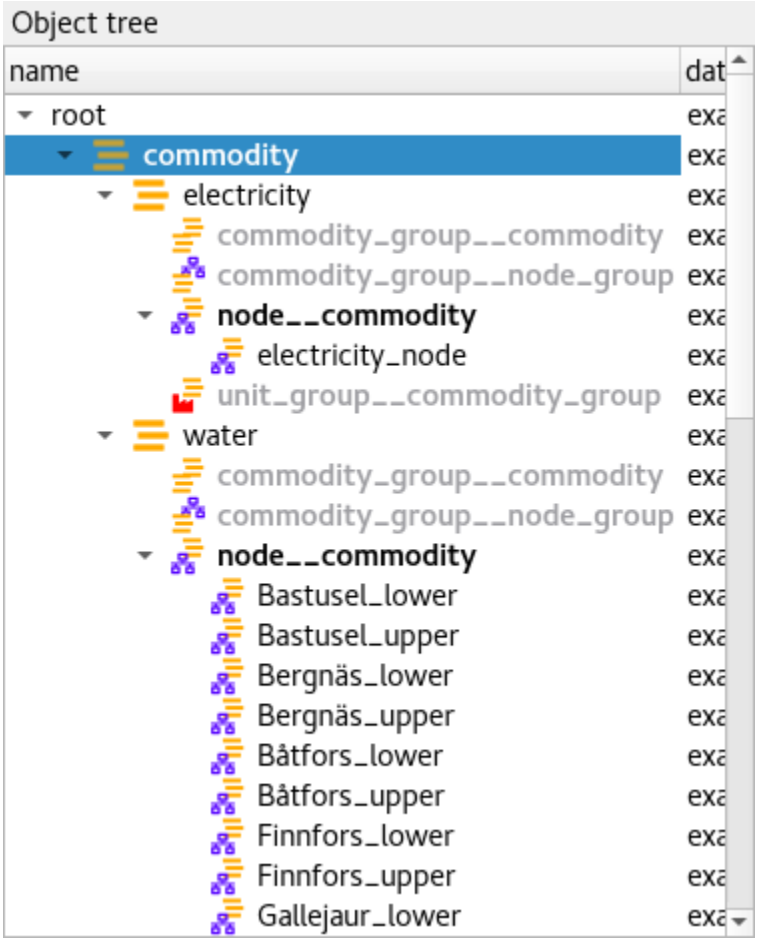
11.3 Viewing data

This section describes the available tools to view data.

11.3.1 Viewing entities and classes

Using *Entity trees*

Entity trees present the structure of classes and entities in all databases in the shape of a tree:



In *Object tree*:

- To view all object classes from all databases, expand the root item (automatically expanded when loading the form).
- To view all objects of a class, expand the corresponding object class item.
- To view all relationship classes involving an object class, expand any objects of that class.
- To view all relationships of a class involving a given object, expand the corresponding relationship class item under the corresponding object item.

In *Relationship tree*:

- To view all relationship classes from all databases, expand the root item (automatically expanded when loading the form).
- To view all relationships of a class, expand the corresponding relationship class item.

Note: To expand an item in *Object tree* or *Relationship tree*, double-click on the item or press the right arrow while it's active. Items in gray don't have any children, thus they cannot be expanded. To collapse an expanded item, double-click on it again or press the left arrow while it's active.

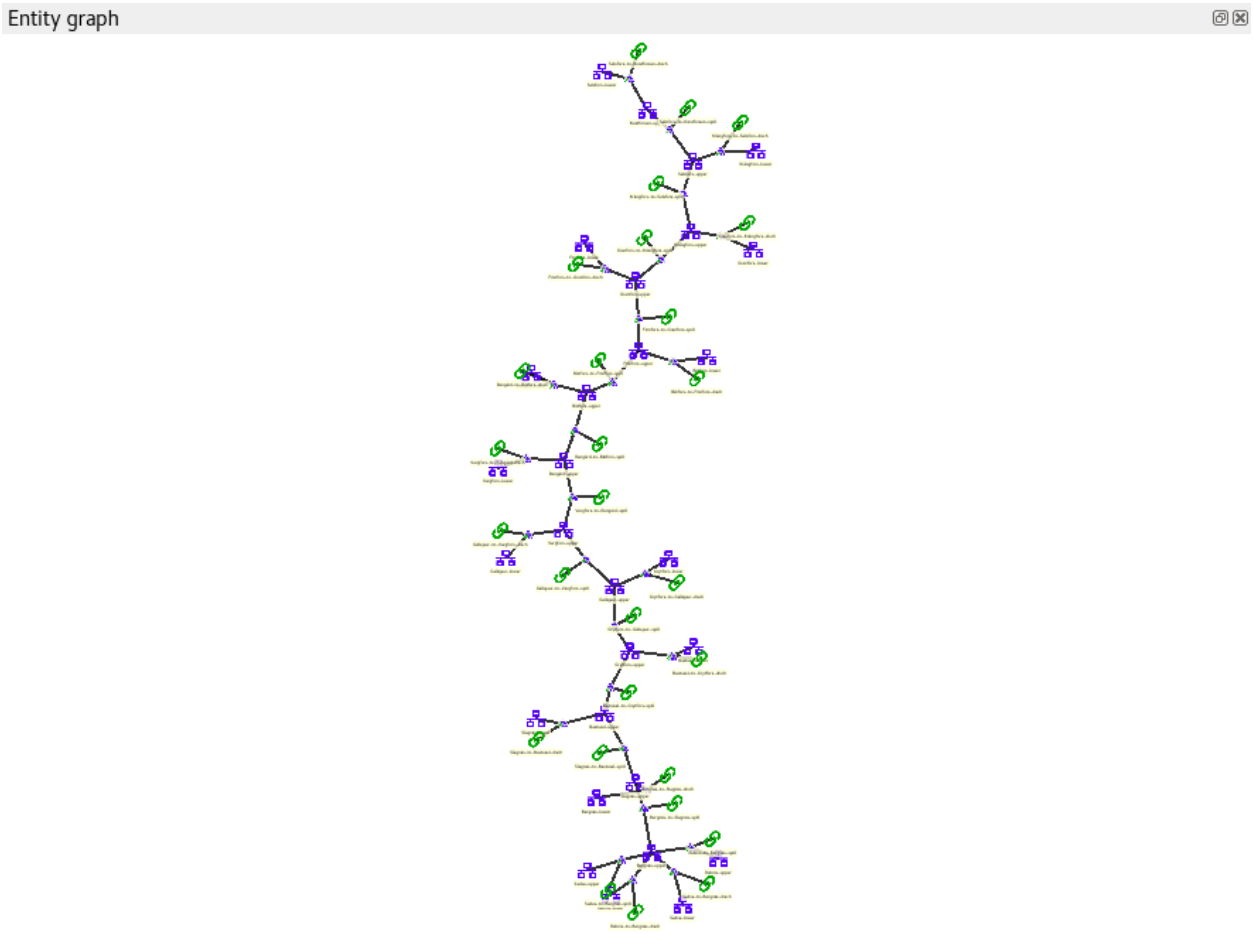
Tip: To expand or collapse an item and all its descentants in *Object tree* or *Relationship tree*, right click on the item

to display the context menu, and select **Fully expand** or **Fully collapse.**

Tip: In *Object tree*, the same relationship appears in many places (as many as it has dimensions). To jump to the next ocurrence of a relationship item, either double-click on the item, or right-click on it to display the context menu, and select **Find next**.

Using *Entity graph*

Entity graph presents the structure of classes and entities from one database in the shape of a graph:



Tip: To see it in action, check out this video.

Building the graph

To build the graph, select any number of items in either *Object tree* or *Relationship tree*. What is included in the graph depends on the specific selection you make:

- To include all objects and relationships from the database, select the root item in either *Object tree* or *Relationship tree*.
- To include all objects of a class, select the corresponding class item in *Object tree*.
- To include all relationships of a class, select the corresponding class item in *Relationship tree*.
- To include all relationships of a specific class involving a specific object, select the corresponding relationship class item under the corresponding object item in *Object tree*.
- To include specific objects or relationships, select the corresponding item in either *Object tree* or *Relationship tree*.

Note: In *Entity graph*, a small unnamed vertex represents a relationship, whereas a bigger named vertex represents an object. An arc between a relationship and an object indicates that the object is a member in that relationship.

The graph automatically includes relationships whenever *all* the member objects are included (even if these relationships are not selected in *Object tree* or *Relationship tree*). You can change this behavior to automatically include relationships whenever *any* of the member objects are included. To do this, enable **Auto-expand objects** via the **Graph** menu, or via *Entity graph*'s context menu.

Tip: To *extend* the selection in *Object tree* or *Relationship tree*, press and hold the **Ctrl** key while clicking on the items.

Tip: *Object tree* and *Relationship tree* also support **Sticky selection**, which allows one to extend the selection by clicking on items *without pressing Ctrl*. To enable **Sticky selection**, select **Settings** from the hamburger menu, and check the corresponding box.

Manipulating the graph

You can move items in the graph by dragging them with your mouse. By default, each items moves individually. To make relationship items move along with their member objects, select **Settings** from the hamburger menu and check the box next to, *Move relationships along with objects in Entity graph*.

To display *Entity graph*'s context menu, just right-click on an empty space in the graph.

- To save the position of items into the database, select the items in the graph and choose **Save positions** from the context menu. To clear saved positions, select the items again and choose **Clear saved positions** from the context menu.
- To hide part of the graph, select the items you want to hide and choose **Hide** from context menu. To show the hidden items again, select **Show hidden** from the context menu.
- To prune the graph, select the items you want to prune and then choose **Prune entities** or **Prune classes** from the context menu. To restore specific prunned items, display the context menu, hover **Restore** and select the items you want to restore from the popup menu. To restore all prunned items at once, select **Restore all** from the context menu.
- To zoom in and out, scroll your mouse wheel over *Entity graph* or use **Zoom** buttons in the context menu.

- To rotate clockwise or anti-clockwise, press and hold the **Shift** key while scrolling your mouse wheel, or use the **Rotate** buttons in the context menu.
- To adjust the arcs' lenght, use the **Arc length** buttons in the context menu.
- To rebuild the graph after moving items around, select **Rebuild graph** from the context menu.
- To export the current graph as a PDF file, select **Export graph as PDF** from the context menu.

Note: *Entity graph* supports extended selection and rubber-band selection. To extend a selection, press and hold **Ctrl** while clicking on the items. To perform rubber-band selection, press and hold **Ctrl** while dragging your mouse around the items you want to select.

Note: Prunned items are remembered across graph builds.

To display an object or relationship item's context menu, just right-click on it.

- To expand or collapse relationships for an object item, hover **Expand** or **Collapse** and select the relationship class from the popup menu.

11.3.2 Viewing parameter definitions and values

Using *Stacked tables*

Stacked tables present object and relationship parameter data from all databases in the form of stacked tables:

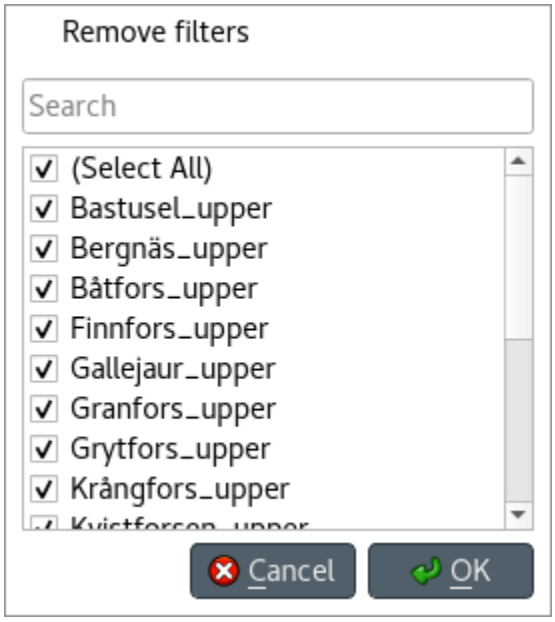
Object parameter value

object_class_name	object_name	parameter_name	value	database
model	instance	duration_unit	hour	example
model	instance	model_end	2019-01-08 00:00:00	example
model	instance	model_start	2019-01-01 00:00:00	example
node	Bastusel_upper	demand	-0.2579768519	example
node	Bastusel_upper	fix_node_state	Time series	example
node	Bastusel_upper	has_state	value_true	example
node	Bastusel_upper	node_state_cap	8208.0	example
node	Bergnäs_upper	demand	-22.29	example
node	Bergnäs_upper	fix_node_state	Time series	example
node	Bergnäs_upper	has_state	value_true	example
node	Bergnäs_upper	node_state_cap	216120.0	example
node	Båtfors_upper	demand	-2.0	example
node	Båtfors_upper	fix_node_state	Time series	example
node	Båtfors_upper	has_state	value_true	example
node	Båtfors_upper	node_state_cap	1330.0	example
node	Finnfors_upper	demand	0.0	example
node	Finnfors_upper	fix_node_state	Time series	example
node	Finnfors_upper	has_state	value_true	example

To filter *Stacked tables* by any entities and/or classes, select the corresponding items in either *Object tree*, *Relationship tree*, or *Entity graph*. To remove all these filters, select the root item in either *Object tree* or *Relationship tree*.

To filter parameter definitions and values by certain parameter tags, select those tags in *Parameter tag toolbar*.

To apply a custom filter on a *Stacked table*, click on any horizontal header. A menu will pop up listing the items in the corresponding column:



Uncheck the items you don't want to see in the table and press **Ok**. Additionally, you can type in the search bar at the top of the menu to filter the list of items. To remove the current filter, select **Remove filters**.

To filter a *Stacked table* according to a selection of items in the table itself, right-click on the selection to show the context menu, and then select **Filter by** or **Filter excluding**. To remove these filters, select **Remove filters** from the header menus of the filtered columns.

Tip: You can rearrange columns in *Stacked tables* by dragging the headers with your mouse. The ordering will be remembered the next time you open Spine DB editor.

11.3.3 Viewing parameter values and relationships

Using *Pivot table* and *Frozen table*

Pivot table and *Frozen table* present data for an individual class from one database in the form of a pivot table, optionally with frozen dimensions:

Pivot table

			parameter	connection_...	fix
	connection	node1	node2		
	Bastusel_to_Grytfors_disch	Grytfors_upper	Bastusel_lower	1h	
	Bastusel_to_Grytfors_spill	Grytfors_upper	Bastusel_upper	150m	
	Bergnäs_to_Slagnäs_disch	Slagnäs_upper	Bergnäs_lower	1h	
	Bergnäs_to_Slagnäs_spill	Slagnäs_upper	Bergnäs_upper	1h	
	Båtfors_to_Finnfors_disch	Finnfors_upper	Båtfors_lower	3h	
	Båtfors_to_Finnfors_spill	Finnfors_upper	Båtfors_upper	3h	
	Finnfors_to_Granfors_disch	Granfors_upper	Finnfors_lower	3h	
	Finnfors_to_Granfors_spill	Granfors_upper	Finnfors_upper	3h	
	Gallejaur_to_Vargfors_disch	Vargfors_upper	Gallejaur_lower	30m	
	Gallejaur_to_Vargfors_spill	Vargfors_upper	Gallejaur_upper	150m	
	Granfors_to_Krångfors_disch	Krångfors_upper	Granfors_lower	3h	
	Granfors_to_Krångfors_spill	Krångfors_upper	Granfors_upper	3h	
	Grytfors_to_Gallejaur_disch	Gallejaur_upper	Grytfors_lower	15m	
	Grytfors_to_Gallejaur_spill	Gallejaur_upper	Grytfors_upper	15m	
	Krångfors_to_Selsfors_disch	Selsfors_upper	Krångfors_lower	3h	
	Krångfors_to_Selsfors_spill	Selsfors_upper	Krångfors_upper	3h	
	Rebnis_to_Bergnäs_disch	Bergnäs_upper	Rebnis_lower	2D	
	Rebnis_to_Bergnäs_spill	Bergnäs_upper	Rebnis_upper	2D	
	Rengård_to_Båtfors_disch	Båtfors_upper	Rengård_lower	3h	
	Rengård_to_Båtfors_spill	Båtfors_upper	Rengård_upper	3h	
	Sadva_to_Bergnäs_disch	Bergnäs_upper	Sadva_lower	2D	
	Sadva_to_Bergnäs_spill	Bergnäs_upper	Sadva_upper	2D	
	Selsfors_to_Kvistforsen_disch	Kvistforsen_upper	Selsfors_lower	3h	
	Selsfors_to_Kvistforsen_spill	Kvistforsen_upper	Selsfors_upper	3h	
	Slagnäs_to_Bastusel_disch	Bastusel_upper	Slagnäs_lower	4h	
	Slagnäs_to_Bastusel_spill	Bastusel_upper	Slagnäs_upper	4h	
	Vargfors_to_Rengård_disch	Rengård_upper	Vargfors_lower	3h	

To populate the tables with data for a certain class, just select the corresponding class item in either *Object tree* or *Relationship tree*.

Selecting the input type

Pivot table and *Frozen table* support four different input types:

- **Parameter value** (the default): it shows objects, parameter definitions, alternatives, and databases in the headers, and corresponding parameter values in the table body.
- **Index expansion**: Similar to the above, but it also shows parameter indexes in the headers. Indexes are extracted from special parameter values, such as time-series.
- **Relationship**: it shows objects, and databases in the headers, and corresponding relationships in the table body. It only works when selecting a relationship class in *Relationship tree*.
- **Scenario**: it shows scenarios, alternatives, and databases in the header, and corresponding *rank* in the table body.

You can select the input type from the **Pivot** section in the hamburger menu.

Note: In *Pivot table*, header blocks in the top-left area indicate what is shown in each horizontal and vertical header. For example, in **Parameter value** input type, by default, the horizontal header has two rows, listing alternative and parameter names, respectively; whereas the vertical header has one or more columns listing object names.

Pivoting and freezing

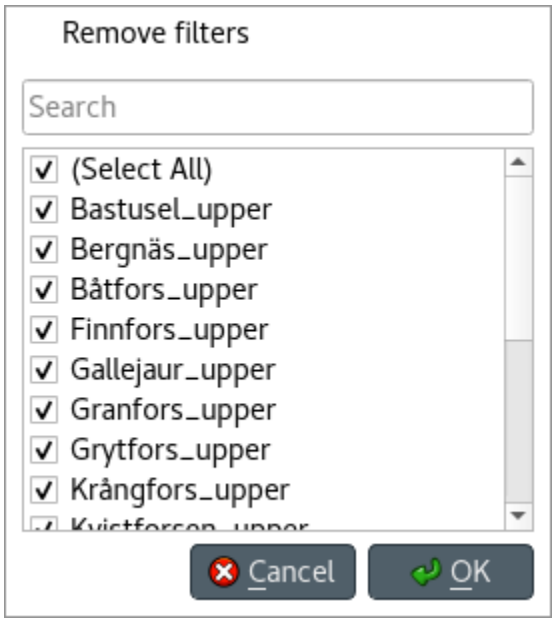
To pivot the data, drag a header block across the top-left area of the table. You can turn a horizontal header into a vertical header and viceversa, as well as rearrange headers vertically or horizontally.

To freeze a dimension, drag the corresponding header block from *Pivot table* into *Frozen table*. To unfreeze a frozen dimension, just do the opposite.

Note: Your pivoting and freezing selections for any class will be remembered when switching to another class.

Filtering

To apply a custom filter on *Pivot table*, click on the arrow next to the name of any header block. A menu will pop up listing the items in the corresponding row or column:

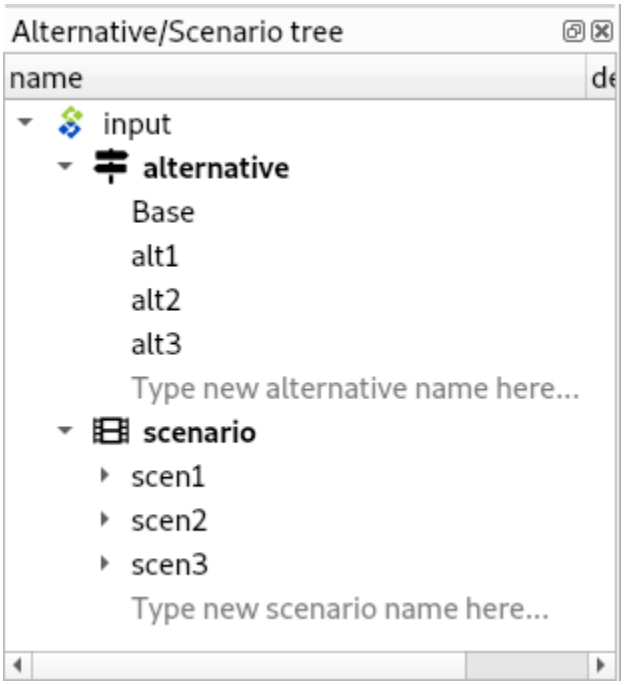


Uncheck the items you don't want to see in the table and press **Ok**. Additionally, you can type in the search bar at the top of the menu to filter the list of items. To remove the current filter, select **Remove filters**.

To filter the pivot table by an individual vector across the frozen dimensions, select the corresponding row in *Frozen table*.

11.3.4 Viewing alternatives and scenarios

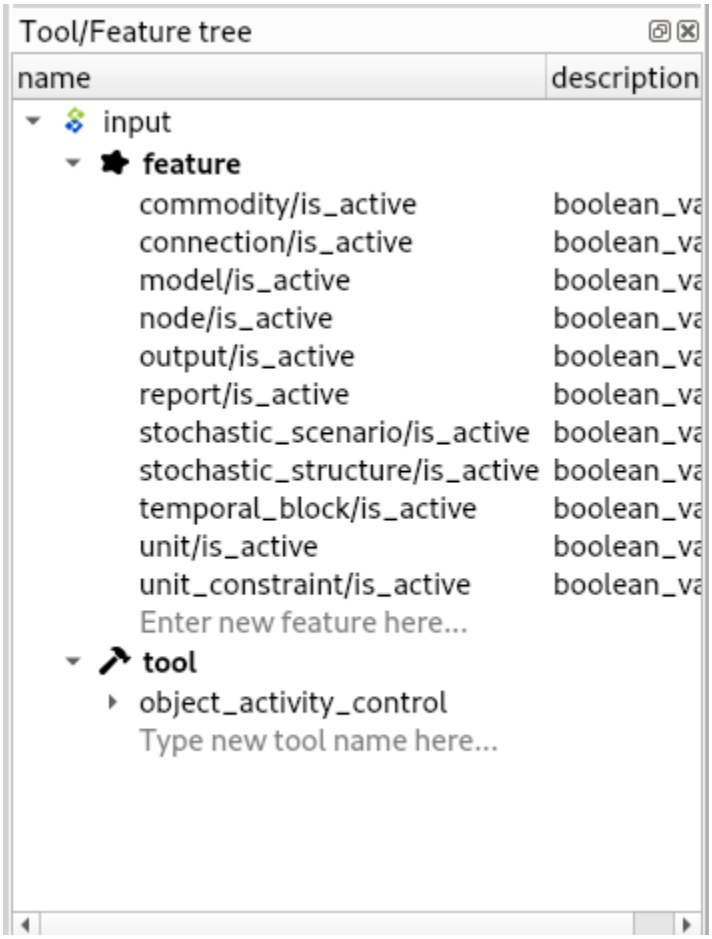
You can find alternatives and scenarios from all databases under *Alternative/Scenario tree*:



To view the alternatives and scenarios from each database, expand the root item for that database. To view all alternatives, expand the **alternative** item. To view all scenarios, expand the **scenario** item. To view the alternatives for a particular scenario, expand the **scenario_alternative** item under the corresponding scenario item.

11.3.5 Viewing tools and features

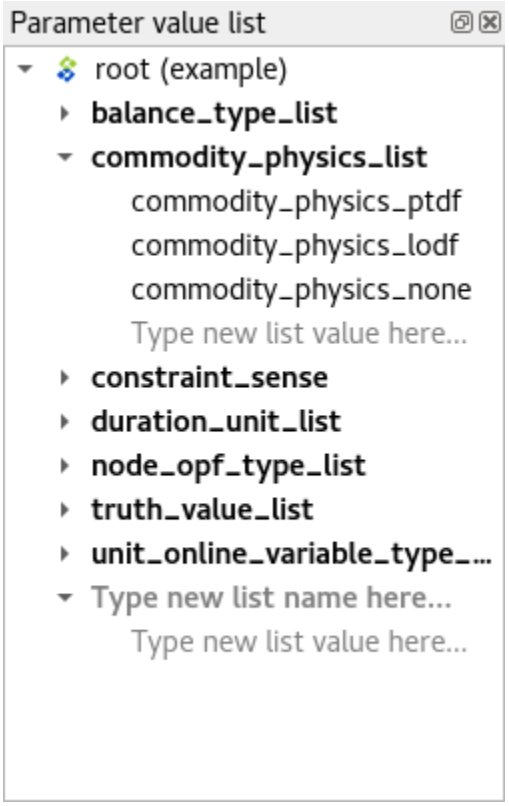
You can find tools, features, and methods from all databases under *Tool/Feature tree*:



To view the features and tools from each database, expand the root item for that database. To view all features, expand the **feature** item. To view all tools, expand the **tool** item. To view the features for a particular tool, expand the **tool_feature** item under the corresponding tool item. To view the methods for a particular tool-feature, expand the **tool_feature_method** item under the corresponding tool-feature item.

11.3.6 Viewing parameter value lists

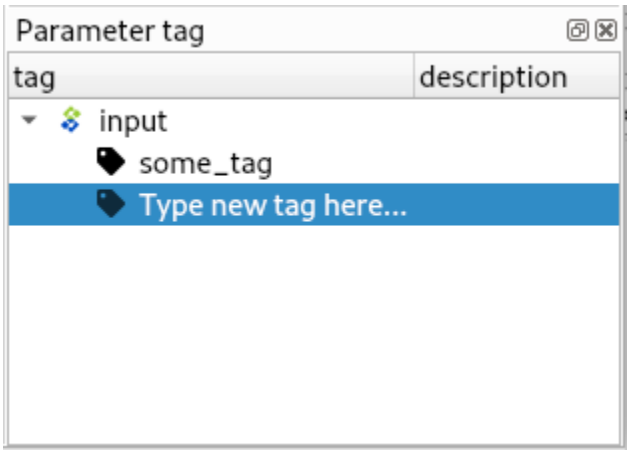
You can find parameter value lists from all databases under *Parameter value list*:



To view the parameter value lists from each database, expand the root item for that database. To view the values for each list, expand the corresponding list item.

11.3.7 Viewing parameter tags

You can find parameter tags from all databases under *Parameter tag*:



To view the tags from each database, expand the root item for that database.

11.4 Adding data

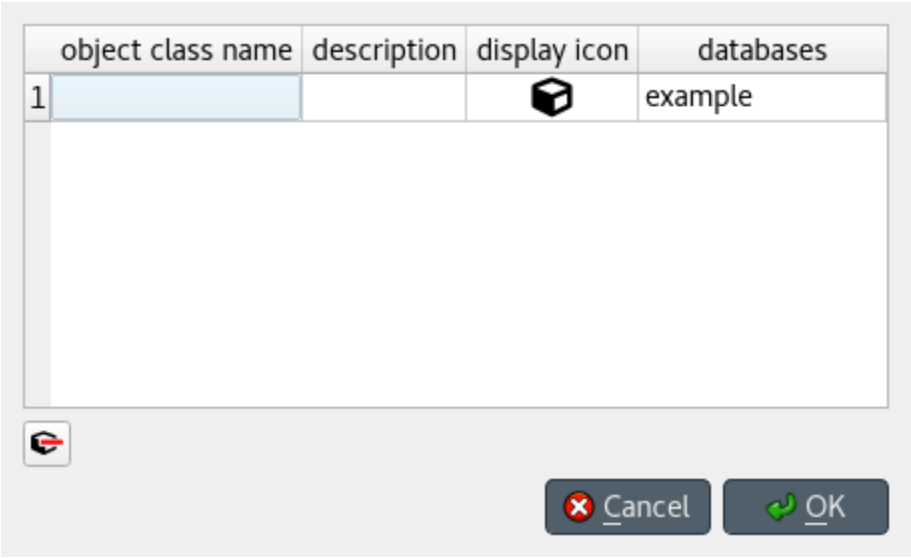
This section describes the available tools to add new data.

11.4.1 Adding object classes

From *Object tree*

Right-click on the root item in *Object tree* to display the context menu, and select **Add object classes**.

The *Add object classes* dialog will pop up:



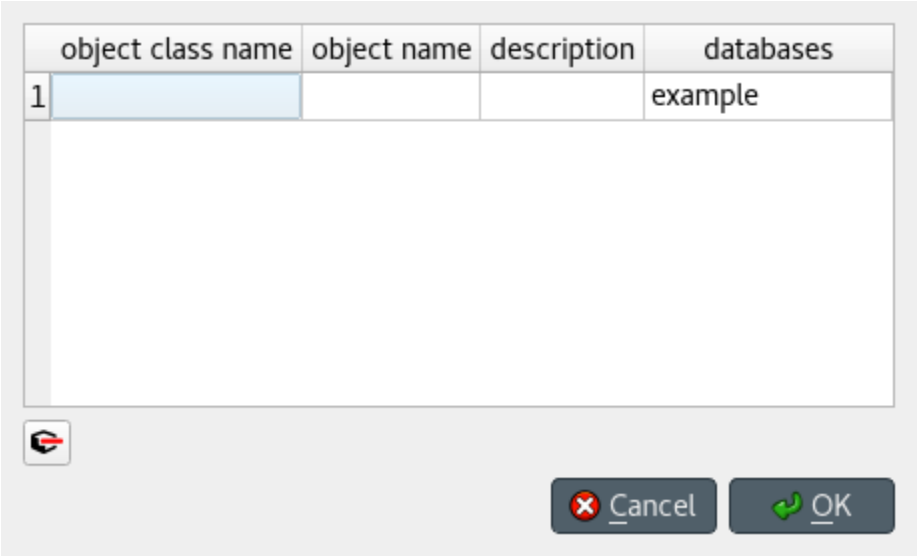
Enter the names of the classes you want to add under the *object class name* column. Optionally, you can enter a description for each class under the *description* column. To select icons for your classes, double click on the corresponding cell under the *display icon* column. Finally, select the databases where you want to add the classes under *databases*. When you're ready, press **Ok**.

11.4.2 Adding objects

From *Object tree* or *Entity graph*

Right-click on an object class item in *Object tree*, or on an empty space in the *Entity graph*, and select **Add objects** from the context menu.

The *Add objects* dialog will pop up:



Enter the names of the object classes under *object class name*, and the names of the objects under *object name*. To display a list of available classes, start typing or double click on any cell under the *object class name* column. Optionally, you can enter a description for each object under the *description* column. Finally, select the databases where you want to add the objects under *databases*. When you're ready, press **Ok**.

From *Pivot table*

To add an object to a specific class, bring the class to *Pivot table* using any input type (see *Using Pivot table and Frozen table*). Then, enter the object name in the last cell of the header corresponding to that class.

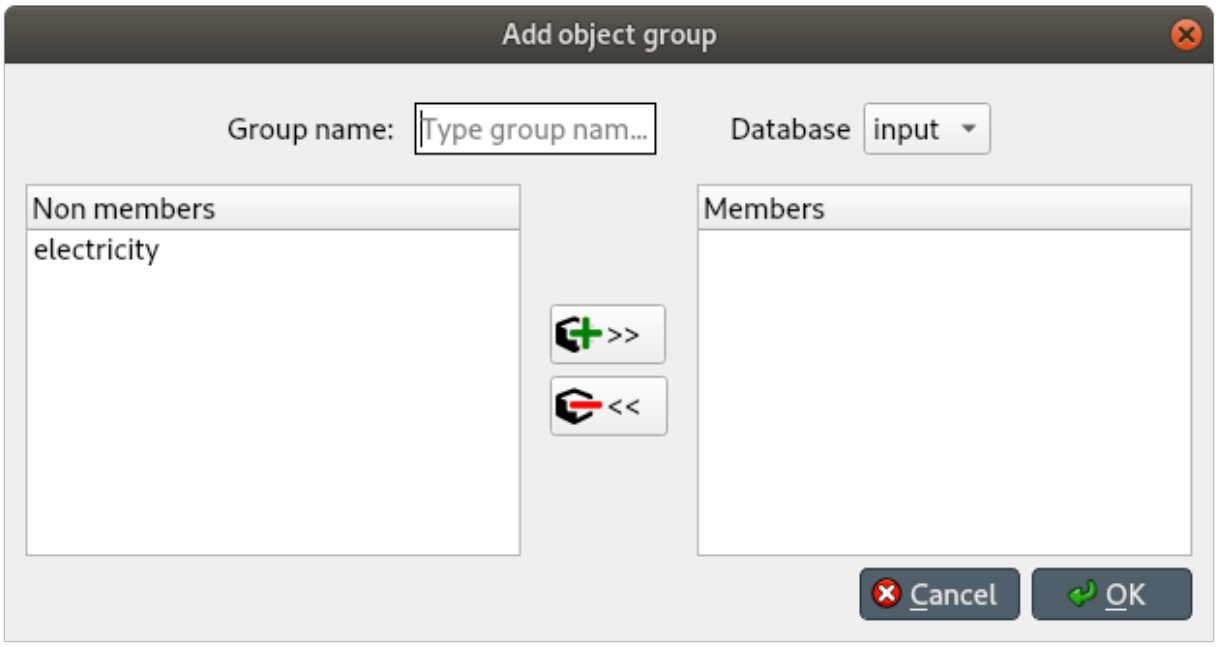
Duplicating objects

To duplicate an existing object with all its relationships and parameter values, right-click over the corresponding object item in *Object tree* to display the context menu, and select **Duplicate object**. Enter a name for the duplicate and press **Ok**.

11.4.3 Adding object groups

Right-click on an object class item in *Object tree*, and select **Add object group** from the context menu.

The *Add object group* dialog will pop up:



Enter the name of the group, adn select the database where you want the group to be created. Select the member objects under *Non members*, and press the button in the middle that has a plus sign. Multiple selection works.

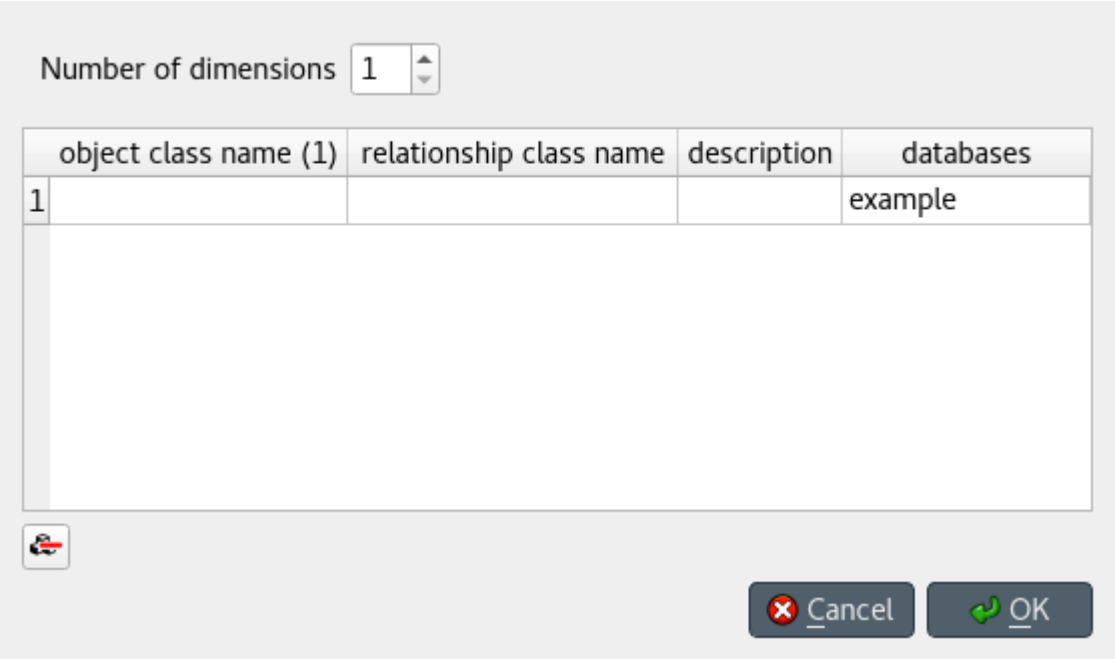
When you're happy with your selections, press **Ok** to add the group to the database.

11.4.4 Adding relationship classes

From *Object tree* or *Relationship tree*

Right-click on an object class item in *Object tree*, or on the root item in *Relationship tree*, and select **Add relationship classes** from the context menu.

The *Add relationship classes* dialog will pop up:



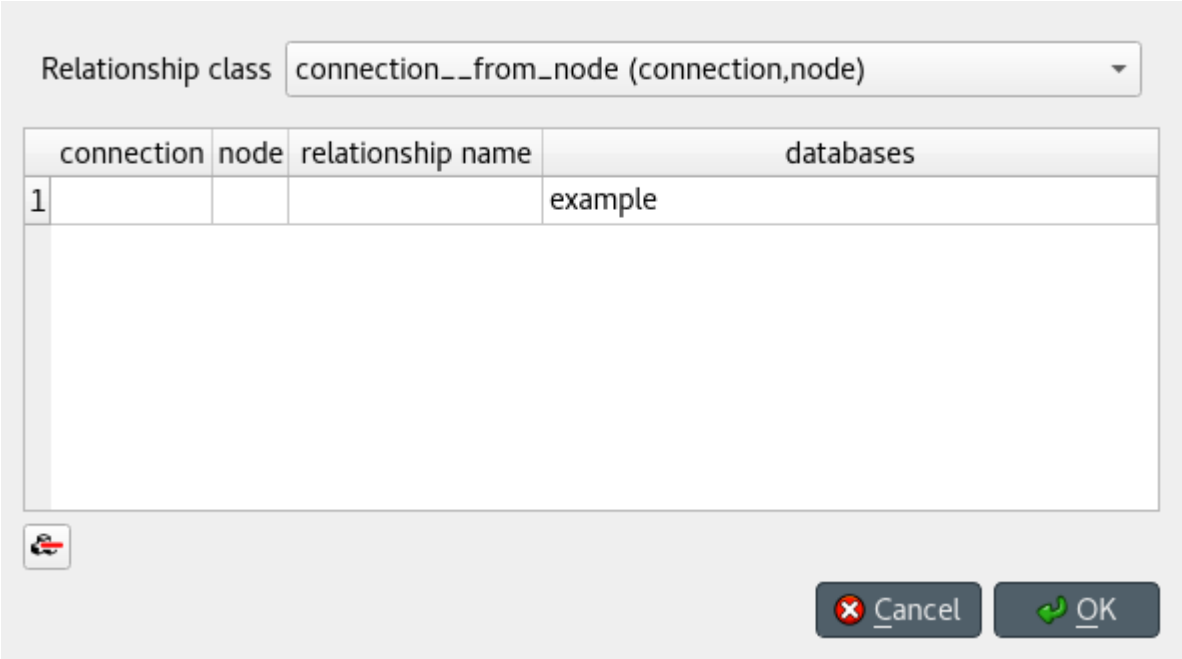
Select the number of dimensions using the spinbox at the top; then, enter the names of the object classes for each dimension under each *object class name* column, and the names of the relationship classes under *relationship class name*. To display a list of available object classes, start typing or double click on any cell under the *object class name* columns. Optionally, you can enter a description for each relationship class under the *description* column. Finally, select the databases where you want to add the relationship classes under *databases*. When you're ready, press **Ok**.

11.4.5 Adding relationships

From *Object tree* or *Relationship tree*

Right-click on a relationship class item either in *Object tree* or *Relationship tree*, and select **Add relationships** from the context menu.

The *Add relationships* dialog will pop up:



Select the relationship class from the combo box at the top; then, enter the names of the objects for each member object class under the corresponding column, and the name of the relationship under *relationship name*. To display a list of available objects for a member class, start typing or double click on any cell under that class's column. Finally, select the databases where you want to add the relationships under *databases*. When you're ready, press **Ok**.

From *Pivot table*

To add a relationship for a specific class, bring the class to *Pivot table* using the **Relationship** input type (see *Using Pivot table and Frozen table*). The *Pivot table* headers will be populated with all possible combinations of objects across the member classes. Locate the objects you want as members in the new relationship, and check the corresponding box in the table body.

From *Entity graph*

Make sure all the objects you want as members in the new relationship are in the graph. To start the relationship, either double click on one of the object items, or right click on it to display the context menu, and choose **Add relationships**. A menu will pop up showing the available relationship classes. Select the class you want; the mouse cursor will adopt a cross-hairs shape. Click on each of the remaining member objects, one by one and in the right order, to add them to the relationship. Once you've added enough objects for the relationship class, a dialog will pop up. Check the boxes next to the relationships you want to add, and press **Ok**.

Tip: All the *Add…* dialogs support pasting tabular (spreadsheet) data from the clipboard. Just select any cell in the table and press **Ctrl+V**. If needed, the table will grow to accommodate the exceeding data. To paste data on multiple cells, select all the cells you want to paste on and press **Ctrl+V**.

11.4.6 Adding parameter definitions

From *Stacked tables*

To add new parameter definitions for an object class, just fill the last empty row of *Object parameter definition*. Enter the name of the class under *object_class_name*, and the name of the parameter under *parameter_name*. To display a list of available object classes, start typing or double click under the *object_class_name* column. Optionally, you can also specify a default value, a parameter value list, or any number of parameter tags under the appropriate columns. The parameter is added when the background of the cells under *object_class_name* and *parameter_name* become gray.

To add new parameter definitions for a relationship class, just fill the last empty row of *Relationship parameter definition*, following the same guidelines as above.

From *Pivot table*

To add a new parameter definition for a class, bring the corresponding class to *Pivot table* using the **Parameter value** input type (see *Using Pivot table and Frozen table*). The **parameter** header of *Pivot table* will be populated with existing parameter definitions for the class. Enter a name for the new parameter in the last cell of that header.

11.4.7 Adding parameter values

From *Stacked tables*

To add new parameter values for an object, just fill the last empty row of *Object parameter value*. Enter the name of the class under *object_class_name*, the name of the object under *object_name*, the name of the parameter under *parameter_name*, and the name of the alternative under *alternative_name*. Optionally, you can also specify the parameter value right away under the *value* column. To display a list of available object classes, objects, parameters, or alternatives, just start typing or double click under the appropriate column. The parameter value is added when the background of the cells under *object_class_name*, *object_name*, and *parameter_name* become gray.

To add new parameter values for a relationship class, just fill the last empty row of *Relationship parameter value*, following the same guidelines as above.

Note: To add parameter values for an object, the object has to exist beforehand. However, when adding parameter values for a relationship, you can specify any valid combination of objects under *object_name_list*, and a relationship will be created among those objects if one doesn't yet exist.

From *Pivot table*

To add parameter value for any object or relationship, bring the corresponding class to *Pivot table* using the **Parameter value** input type (see *Using Pivot table and Frozen table*). Then, enter the parameter value in the corresponding cell in the table body.

Tip: All *Stacked tables* and *Pivot table* support pasting tabular (e.g., spreadsheet) data from the clipboard. Just select any cell in the table and press **Ctrl+V**. If needed, *Stacked tables* will grow to accommodate the exceeding data. To paste data on multiple cells, select all the cells you want to paste on and press **Ctrl+V**.

11.4.8 Adding tools, features, and methods

To add a new feature, go to *Tool/Feature tree* and select the last item under **feature** in the appropriate database, start typing or press **F2** to display available parameter definitions, and select the one you want to become a feature.

Note: Only parameter definitions that have associated a parameter value list can become features.

To add a new tool, just select the last item under **tool** in the appropriate database, and enter the name of the tool.

To add a feature for a particular tool, drag the feature item and drop it over the **tool_feature** list under the corresponding tool.

To add a new method for a tool-feature, select the last item under *tool_feature_method* (in the appropriate database), start typing or press **F2** to display available methods, and select the one you want to add.

11.4.9 Adding alternatives and scenarios

From *Alternative/Scenario tree*

To add a new alternative, just select the last item under **alternative** in the appropriate database, and enter the name of the alternative.

To add a new scenario, just select the last item under **scenario** in the appropriate database, and enter the name of the scenario.

To add an alternative for a particular scenario, drag the alternative item and drop it over the **scenario_alternative** list under the corresponding scenario. The position where you drop it determines the alternative's *rank* within the scenario.

Note: Alternatives with higher rank have priority when determining the parameter value for a certain scenario. If the parameter value is specified for two alternatives, and both of them happen to coexist in a same scenario, the value from the alternative with the higher rank is picked.

From *Pivot table*

Select the **Scenario** input type (see *Using Pivot table and Frozen table*). To add a new scenario, enter a name in the last cell of the **scenario** header. To add a new alternative, enter a name in the last cell of the **alternative** header.

11.4.10 Adding parameter value lists

To add a new parameter value list, go to *Parameter value list* and select the last item under the appropriate database, and enter the name of the list.

To add new values for the list, select the last empty item under the corresponding list item, and enter the value. To enter a complex value, right-click on the empty item and select **Open editor** from the context menu.

Note: To be actually added to the database, a parameter value list must have at least one value.

11.4.11 Adding parameter tags

To add a new parameter tag, go to *Parameter tag* and select the last item under the appropriate database, and enter the tag's name.

11.5 Updating data

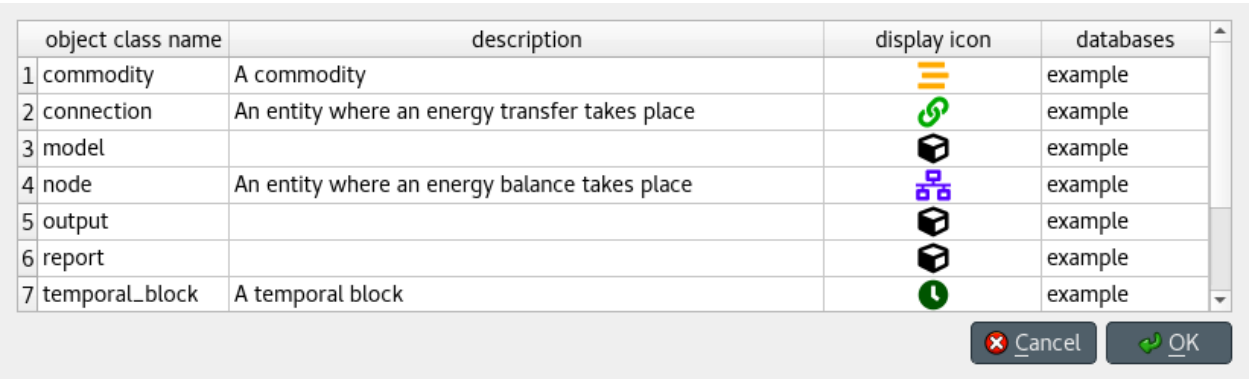
This section describes the available tools to update existing data.

11.5.1 Updating entities and classes

From *Object tree*, *Relationship tree*, or *Entity graph*

Select any number of entity and/or class items in *Object tree* or *Relationship tree*, or any number of object and/or relationship items in *Entity graph*. Then, right-click on the selection and choose **Edit…** from the context menu.

One separate *Edit…* dialog will pop up for each selected entity or class type, and the tables will be filled with the current data of selected items. E.g.:



	object class name	description	display icon	databases
1	commodity	A commodity		example
2	connection	An entity where an energy transfer takes place		example
3	model			example
4	node	An entity where an energy balance takes place		example
5	output			example
6	report			example
7	temporal_block	A temporal block		example

Cancel OK

Modify the field(s) you want under the corresponding column(s). Specify the databases where you want to update each item under the *databases* column. When you're ready, press **Ok**.

From *Pivot table*

To rename an object of a specific class, bring the class to *Pivot table* using any input type (see *Using Pivot table and Frozen table*). Then, just edit the appropriate cell in the corresponding class header.

11.5.2 Updating parameter definitions and values

From *Stacked tables*

To update parameter data, just go to the appropriate *Stacked table* and edit the corresponding row.

From *Pivot table*

To rename parameter definitions for a class, bring the corresponding class to *Pivot table* using the **Parameter value** input type (see *Using Pivot table and Frozen table*). Then, just edit the appropriate cell in the **parameter** header.

To modify parameter values for an object or relationship, bring the corresponding class to *Pivot table* using the **Parameter value** input type (see *Using Pivot table and Frozen table*). Then, just edit the appropriate cell in the table body.

11.5.3 Updating alternatives and scenarios

From *Pivot table*

Select the **Scenario** input type (see *Using Pivot table and Frozen table*). To rename a scenario, just edit the proper cell in the **scenario** header. To rename an alternative, just edit the proper cell in the **alternative** header.

From *Alternative/Scenario tree*

To rename a scenario or alternative, just edit the appropriate item in *Alternative/Scenario tree*. To change scenario alternative ranks, just drag and drop the items under **scenario_alternatives**.

11.5.4 Updating tools and features

To change a feature or method, or rename a tool, just edit the appropriate item in *Tool/Feature tree*.

11.5.5 Updating parameter value lists

To rename a parameter value list or change any of its values, just edit the appropriate item in *Parameter value list*.

11.6 Removing data

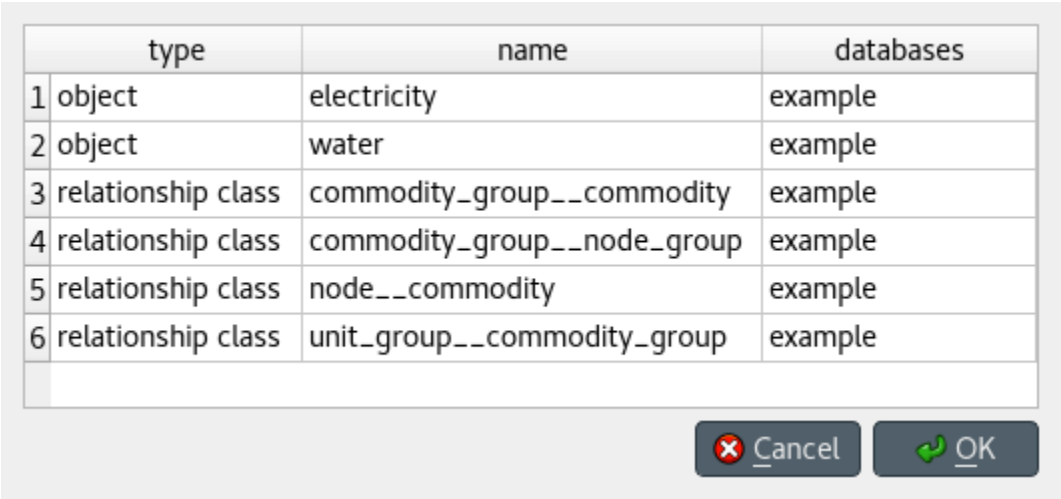
This section describes the available tools to remove data.

11.6.1 Removing entities and classes

From *Object tree*, *Relationship tree*, or *Entity graph*

Select the items in *Object tree*, *Relationship tree*, or *Entity graph*, corresponding to the entities and classes you want to remove. Then, right-click on the selection and choose **Remove** from the context menu.

The *Remove items* dialog will popup:



	type	name	databases
1	object	electricity	example
2	object	water	example
3	relationship class	commodity_group__commodity	example
4	relationship class	commodity_group__node_group	example
5	relationship class	node__commodity	example
6	relationship class	unit_group__commodity_group	example

Cancel OK

Specify the databases from where you want to remove each item under the *databases* column, and press **Ok**.

From *Pivot table*

To remove objects or relationships from a specific class, bring the class to *Pivot table* using the **Parameter value** input type (see *Using Pivot table and Frozen table*), and select the cells in the table headers corresponding to the objects and/or relationships you want to remove. Then, right-click on the selection and choose the corresponding **Remove** option from the context menu.

Alternatively, to remove relationships for a specific class, bring the class to *Pivot table* using the **Relationship** input type (see *Using Pivot table and Frozen table*). The *Pivot table* headers will be populated with all possible combinations of objects across the member classes. Locate the member objects of the relationship you want to remove, and uncheck the corresponding box in the table body.

11.6.2 Removing parameter definitions and values

From *Stacked tables*

To remove parameter definitions or values, go to the relevant *Stacked table* and select any cell in the row corresponding to the items you want to remove Then, right-click on the selection and choose the appropriate **Remove** option from the context menu.

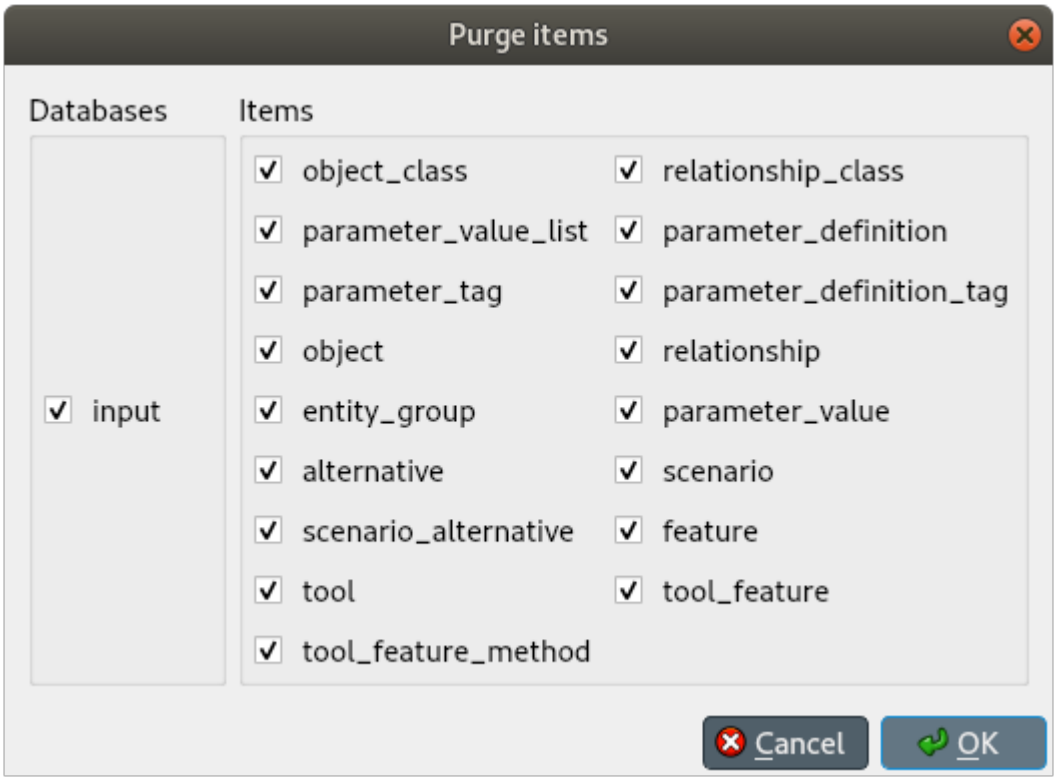
From *Pivot table*

To remove parameter definitions and/or values for a certain class, bring the corresponding class to *Pivot table* using the **Parameter value** input type (see *Using Pivot table and Frozen table*). Then:

1. Select the cells in the *parameter* header corresponding to the parameter definitions you want to remove, right-click on the selection and choose **Remove parameter definitions** from the context menu
2. Select the cells in the table body corresponding to the parameter values you want to remove, right-click on the selection and choose **Remove parameter values** from the context menu.

11.6.3 Purging items

To remove all items of specific types, select **Edit -> Purge** from the hamburger menu. The *Purge items* dialog will pop up:



Select the databases from where you want to remove the items under *Databases*, and the type of items you want to remove under *Items*. Then, press **Ok**.

11.6.4 Removing alternatives and scenarios

From *Pivot table*

Select the **Scenario** input type (see *Using Pivot table and Frozen table*). To remove scenarios, just select the proper cells in the **scenario** header, right-click on the selection and chose **Remove** from the context menu. To remove alternatives, just edit the proper cells in the **alternative** header, right-click on the selection and chose **Remove** from the context menu.

From *Alternative/Scenario tree*

To remove a scenario or alternative, just select the correspoding items in *Alternative/Scenario tree*, right-click on the selection and chose **Remove** from the context menu.

11.6.5 Removing tools and features

To remove a feature, tool, or method, just select the correspoding items in *Tool/Feature tree*, right-click on the selection and chose **Remove** from the context menu.

11.6.6 Removing parameter value lists

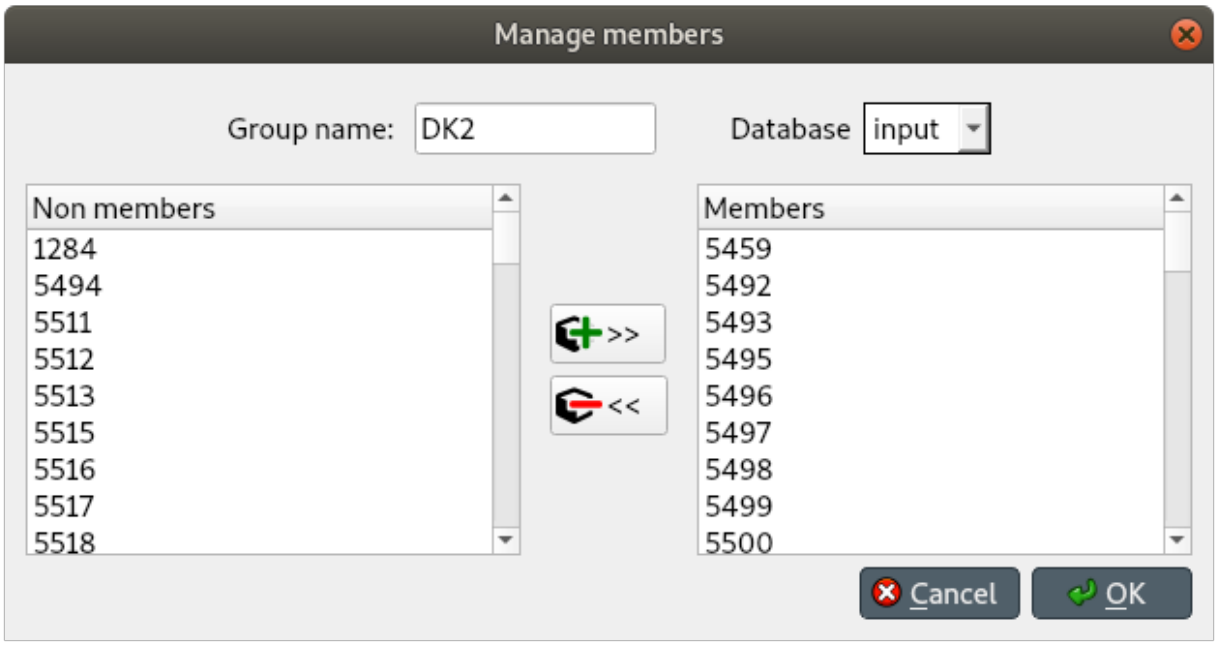
To remove a parameter value list or any of its values, just select the correspoding items in *Parameter value list*, right-click on the selection and chose **Remove** from the context menu.

11.7 Managing data

This section describes the available tools to manage data, i.e., adding, updating or removing data at the same time.

11.7.1 Managing object groups

To modify object groups, expand the corresponding item in *Object tree* to display the **members** item, right-click on the latter and select **Manage members** from the context menu. The *Manage parameter tags* dialog will pop up:



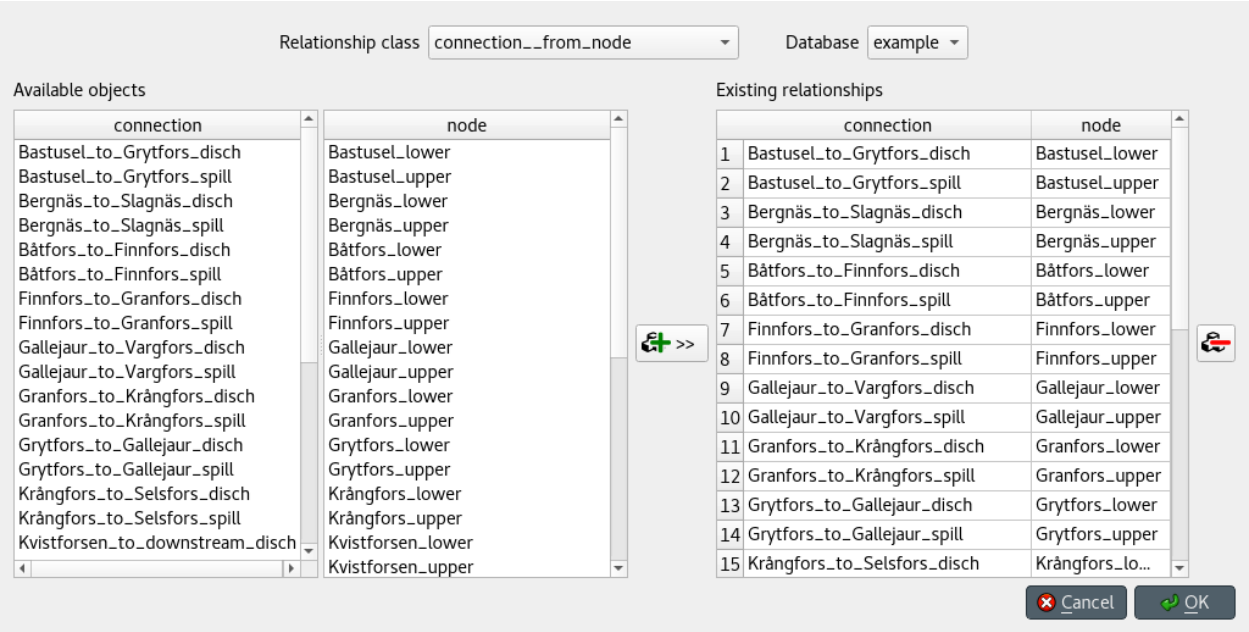
To add new member objects, select them under *Non members*, and press the button in the middle that has a plus sign. To remove current member objects, select them under *Members*, and press the button in the middle that has a minus sign. Multiple selection works in both lists.

When you're happy, press **Ok**.

Note: Changes made using the *Manage members* dialog are not applied to the database until you press **Ok**.

11.7.2 Managing relationships

Select **Edit -> Manage relationships** from the menu bar. The *Manage relationships* dialog will pop up:



To get started, select a relationship class and a database from the combo boxes at the top.

To add relationships, select the member objects for each class under *Available objects* and press the **Add relationships** button at the middle of the form. The relationships will appear at the top of the table under *Existing relationships*.

To add multiple relationships at the same time, select multiple objects for one or more of the classes.

Tip: To *extend* the selection of objects for a class, press and hold the **Ctrl** key while clicking on more items.

Note: The set of relationships to add is determined by applying the *product* operation over the objects selected for each class.

To remove relationships, select the appropriate rows under *Existing relationships* and press the **Remove relationships** button on the right.

When you're happy with your changes, press **Ok**.

Note: Changes made using the *Manage relationships* dialog are not applied to the database until you press **Ok**.

11.8 Importing and exporting data

This section describes the available tools to import and export data.

11.8.1 Overview

Spine database editor supports importing and exporting data in three different formats: SQLite, JSON, and Excel. The SQLite import/export uses the Spine database format. The JSON and Excel import/export use a specific format described below.

Tip: To create a template file with the JSON or Excel format you can simply export an existing Spine database into one of those formats.

Excel format

The Excel format consists of one sheet per object and relationship class. Each sheet can have one of four different formats:

1. Object class with scalar parameter data:

	A	B	C	D	E	F	G
1	sheet_type	entity					
2	entity_type	object					
3	class_name	node					
4	entity_dim_count	1					
5	value_type	single_value					
6	index_dim_count	0					
7							
8	node	alternative	demand	has_state	node_state_cap	state_coeff	
9	Bastusel_upper	Base	-0.2579768519	1	8208	1	
10	Bergnäs_upper	Base	-22.29	1	216120	1	
11	Båtfors_upper	Base	-2	1	1330	1	
12	Finnfors_upper	Base	0	1	300	1	
13	Gallejaur_upper	Base	15.356962963	1	3600	1	
14	Granfors_upper	Base	0	1	280	1	
15	Grytfors_upper	Base	-3.78	1	1248	1	
16	Krångfors_upper	Base	0	1	330	1	
17	Kvistforsen_upper	Base	-1.3273809524	1	1120	1	
18	Rebnis_upper	Base	-3.68	1	205560	1	
19	Rengård_upper	Base	-10.37	1	1400	1	
20	Sadva_upper	Base	-5.43	1	168000	1	
21	Selsfors_upper	Base	0	1	500	1	
22	Slagnäs_upper	Base	0	1	768	1	
23	Vargfors_upper	Base	-3.5584953704	1	4008	1	
24							
25							
26							

2. Object class with indexed parameter data:

	A	B	C	D	E	F
1	sheet_type	entity				
2	entity_type	object				
3	class_name	node				
4	entity_dim_count	1				
5	value_type	time_series				
6	index_dim_count	1				
7						
8	node	Bastusel_upper	Bergnäs_upper	Båtfors_upper	Finnfors_upper	Gallejaur_upper
9	alternative	Base	Base	Base	Base	Base
10	index	fix_node_state	fix_node_state	fix_node_state	fix_node_state	fix_node_state
11	2019-01-01T00:00:00	5581.44	114543.6	1117.2	234	1224
12	2019-01-01T01:00:00	nan	nan	nan	nan	nan
13	2019-01-07T23:00:00	5417.28	105898.8	891.1	234	2808
14						
15						

3. Relationship class with scalar parameter data:

	A	B	C	D	E
1	sheet_type	entity			
2	entity_type	relationship			
3	class_name	connection__node__node			
4	entity_dim_count	3			
5	value_type	single_value			
6	index_dim_count	0			
7					
8	connection	node	node	alternative	connection_flow_delay
9	Bastusel_to_Grytfors_disch	Grytfors_upper	Bastusel_lower	Base	1h
10	Bastusel_to_Grytfors_spill	Grytfors_upper	Bastusel_upper	Base	150m
11	Bergnäs_to_Slagnäs_disch	Slagnäs_upper	Bergnäs_lower	Base	1h
12	Bergnäs_to_Slagnäs_spill	Slagnäs_upper	Bergnäs_upper	Base	1h
13	Båtfors_to_Finnfors_disch	Finnfors_upper	Båtfors_lower	Base	3h
14	Båtfors_to_Finnfors_spill	Finnfors_upper	Båtfors_upper	Base	3h
15	Finnfors_to_Granfors_disch	Granfors_upper	Finnfors_lower	Base	3h
16	Finnfors_to_Granfors_spill	Granfors_upper	Finnfors_upper	Base	3h
17	Gallejaur_to_Vargfors_disch	Vargfors_upper	Gallejaur_lower	Base	30m
18	Gallejaur_to_Vargfors_spill	Vargfors_upper	Gallejaur_upper	Base	150m
19	Granfors_to_Krångfors_disch	Krångfors_upper	Granfors_lower	Base	3h

4. Relationship class with indexed parameter data:

	A	B	C	D	E	F
1	sheet_type	entity				
2	entity_type	relationship				
3	class_name	unit__from_node				
4	entity_dim_count	2				
5	value_type	time_series				
6	index_dim_count	1				
7						
8	unit	unit1	unit2	unit3	unit4	unit5
9	node	electricity_node	electricity_node	gas_node	gas_node	gas_noc
10	alternative	Base	Base	Base	Base	Base
11	index	vom_cost	vom_cost	vom_cost	vom_cost	vom_co
12	2019-01-01T00:00:00	-162.03	-175.56	-207.34	-178.87	-17
13	2019-01-01T01:00:00	-156.36	-283.11	-194.95	-174.71	-17
14	2019-01-01T02:00:00	-151.06	-278.76	-190.41	-168.75	-17
15	2019-01-01T03:00:00	-153.52	-299.57	-185.4	-172.89	-17
16	2019-01-01T04:00:00	-158.91	-285.28	-183.41	-172.13	-22
17	2019-01-01T05:00:00	-164.02	-207.34	-191.54	-171.66	-27
18	2019-01-01T06:00:00	-175.56	-194.95	-202.9	-173.27	-3
19	2019-01-01T07:00:00	-283.11	-190.41	-197.69	-176.97	-40
20						
21						

JSON format

The JSON format consists of a single JSON object with the following `OPTIONAL` keys:

- **object_classes**: the value of this key `MUST` be a JSON array, representing a list of object classes. Each element in this array `MUST` be itself a JSON array and `MUST` have three elements:
  - The first element `MUST` be a JSON string, indicating the object class name.
  - The second element `MUST` be either a JSON string, indicating the object class description, or null.
  - The third element `MUST` be either a JSON integer, indicating the object class icon code, or null.
- **relationship_classes**: the value of this key `MUST` be a JSON array, representing a list of relationships classes. Each element in this array `MUST` be itself a JSON array and `MUST` have three elements:
  - The first element `MUST` be a JSON string, indicating the relationship class name.
  - The second element `MUST` be a JSON array, indicating the member object classes. Each element in this array `MUST` be a JSON string, indicating the object class name.
  - The third element `MUST` be either a JSON string, indicating the relationship class description, or null.
- **parameter_value_lists**: the value of this key `MUST` be a JSON array, representing a list of parameter value lists. Each element in this array `MUST` be itself a JSON array and `MUST` have two elements:
  - The first element `MUST` be a JSON string, indicating the parameter value list name.
  - The second element `MUST` be a JSON array, indicating the values in the list. Each element in this array `MUST` be either a JSON object, string, number, or null, indicating the value.
- **object_parameters**: the value of this key `MUST` be a JSON array, representing a list of object parameter definitions. Each element in this array `MUST` be itself a JSON array and `MUST` have five elements:
  - The first element `MUST` be a JSON string, indicating the object class name.
  - The second element `MUST` be a JSON string, indicating the parameter name.
  - The third element `MUST` be either a JSON object, string, number, or null, indicating the parameter default value.
  - The fourth element `MUST` be a JSON string, indicating the associated parameter value list, or null.
  - The last element `MUST` be either a JSON string, indicating the parameter description, or null.
- **relationship_parameters**: the value of this key `MUST` be a JSON array, representing a list of relationship parameter definitions. Each element in this array `MUST` be itself a JSON array and `MUST` have five elements:
  - The first element `MUST` be a JSON string, indicating the relationship class name.
  - The second element `MUST` be a JSON string, indicating the parameter name.
  - The third element `MUST` be either a JSON object, string, number, or null, indicating the parameter default value.
  - The fourth element `MUST` be a JSON string, indicating the associated parameter value list, or null
  - The last element `MUST` be either a JSON string, indicating the parameter description, or null.
- **objects**: the value of this key `MUST` be a JSON array, representing a list of objects. Each element in this array `MUST` be itself a JSON array and `MUST` have three elements:
  - The first element `MUST` be a JSON string, indicating the object class name.
  - The second element `MUST` be a JSON string, indicating the object name.
  - The third element `MUST` be either a JSON string, indicating the object description, or null.

- **relationships**: the value of this key `MUST` be a JSON array, representing a list of relationships. Each element in this array `MUST` be itself a JSON array and `MUST` have two elements:
  - The first element `MUST` be a JSON string, indicating the relationship class name.
  - The second element `MUST` be a JSON array, indicating the member objects. Each element in this array `MUST` be a JSON string, indicating the object name.
- **object_parameter_values**: the value of this key `MUST` be a JSON array, representing a list of object parameter values. Each element in this array `MUST` be itself a JSON array and `MUST` have four elements:
  - The first element `MUST` be a JSON string, indicating the object class name.
  - The second element `MUST` be a JSON string, indicating the object name.
  - The third element `MUST` be a JSON string, indicating the parameter name.
  - The fourth element `MUST` be either a JSON object, string, number, or null, indicating the parameter value.
- **relationship_parameter_values**: the value of this key `MUST` be a JSON array, representing a list of relationship parameter values. Each element in this array `MUST` be itself a JSON array and `MUST` have four elements:
  - The first element `MUST` be a JSON string, indicating the relationship class name.
  - The second element `MUST` be a JSON array, indicating the relationship's member objects. Each element in this array `MUST` be a JSON string, indicating the object name.
  - The third element `MUST` be a JSON string, indicating the parameter name.
  - The fourth element `MUST` be either a JSON object, string, number, or null, indicating the parameter value.

Example:

```
{
    "object_classes": [
        ["connection", "An entity where an energy transfer takes place",␣
↪280378317271233],
        ["node", "An entity where an energy balance takes place", 280740554077951],
        ["unit", "An entity where an energy conversion process takes place",␣
↪281470681805429],
    ],
    "relationship_classes": [
        ["connection__node__node", ["connection", "node", "node"] , null],
        ["unit__from_node", ["unit", "node"], null],
        ["unit__to_node", ["unit", "node"], null],
    ],
    "parameter_value_lists": [
        ["balance_type_list", ["\"balance_type_node\"", "\"balance_type_group\"", "\
↪"balance_type_none\""]],
        ["truth_value_list", ["\"value_false\"", "\"value_true\""]],
    ],
    "object_parameters": [
        ["connection", "connection_availability_factor", 1.0, null, null],
        ["node", "balance_type", "balance_type_node", "balance_type_list", null],
    ],
    "relationship_parameters": [
        ["connection__node__node", "connection_flow_delay", {"type": "duration", "data":
↪"0h"}, null, null],
        ["unit__from_node", "unit_capacity", null, null, null],
        ["unit__to_node", "unit_capacity", null, null, null],
```

(continues on next page)

(continued from previous page)

```
    ],
    "objects": [
        ["connection", "Bastusel_to_Grytfors_disch", null],
        ["node", "Bastusel_lower", null],
        ["node", "Bastusel_upper", null],
        ["node", "Grytfors_upper", null],
        ["unit", "Bastusel_pwr_plant", null],
    ],
    "relationships": [
        ["connection__node__node", ["Bastusel_to_Grytfors_disch", "Grytfors_upper",
↪"Bastusel_lower"]],
        ["unit__from_node", ["Bastusel_pwr_plant", "Bastusel_upper"]],
        ["unit__to_node", ["Bastusel_pwr_plant", "Bastusel_lower"]],
    ],
    "object_parameter_values": [
        ["node", "Bastusel_upper", "demand", -0.2579768519],
        ["node", "Bastusel_upper", "fix_node_state", {"type": "time_series", "data": {
↪"2018-12-31T23:00:00": 5581.44, "2019-01-07T23:00:00": 5417.28}}],
        ["node", "Bastusel_upper", "has_state", "value_true"],
    ],
    "relationship_parameter_values": [
        ["connection__node__node", ["Bastusel_to_Grytfors_disch", "Grytfors_upper",
↪"Bastusel_lower"], "connection_flow_delay", {"type": "duration", "data": "1h"}],
        ["unit__from_node", ["Bastusel_pwr_plant", "Bastusel_upper"], "unit_capacity",␣
↪127.5],
    ]
}
```

11.8.2 Importing

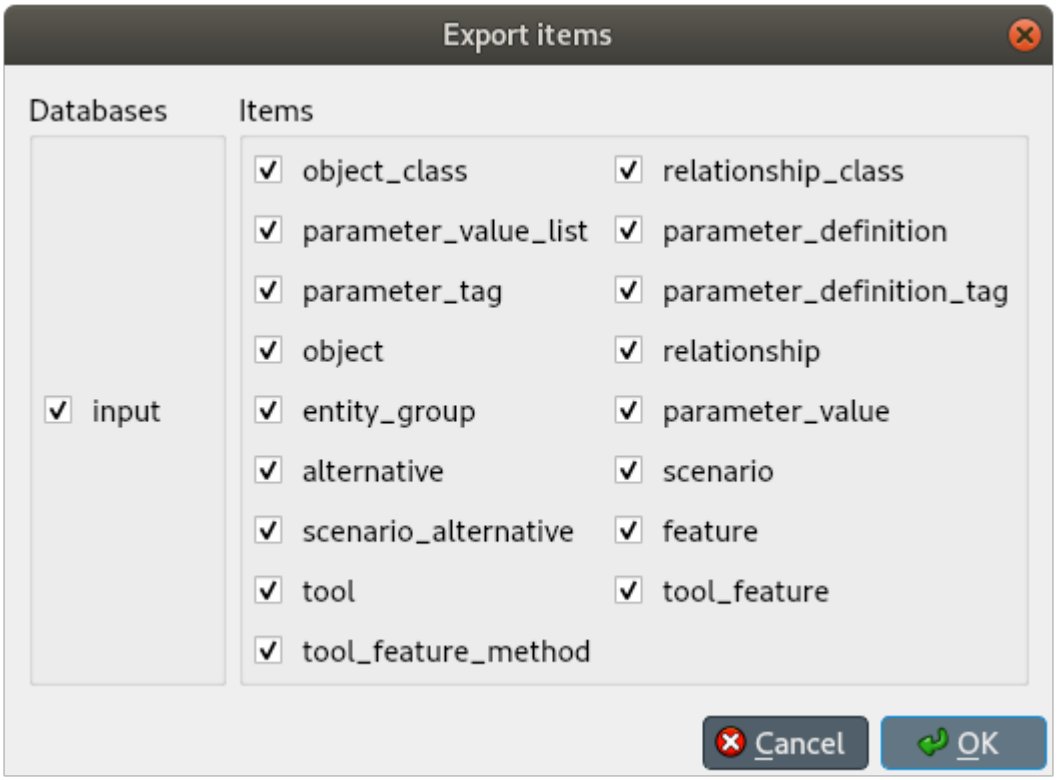
To import a file, select **File –> Import** from the hamburger menu. The *Import file* dialog will pop up. Select the file type (SQLite, JSON, or Excel), enter the path of the file to import, and accept the dialog.

Tip: You can undo import operations using **Edit -> Undo**.

11.8.3 Exporting

Mass export

To export items in mass, select **File –> Export** from the hamburger menu. The *Export items* dialog will pop up:



Select the databases you want to export under *Databases*, and the type of items under *Items*, then press **Ok**. The *Export file* dialog will pop up now. Select the file type (SQLite, JSON, or Excel), enter the path of the file to export, and accept the dialog.

Selective export

To export a specific subset of items, select the corresponding items in either *Object tree* and *Relationship tree*, right click on the selection to bring the context menu, and select **Export**.

The *Export file* dialog will pop up. Select the file type (SQLite, JSON, or Excel), enter the path of the file to export, and accept the dialog.

Session export

To export only uncommitted changes made in the current session, select **File –> Export session** from the hamburger menu.

The *Export file* dialog will pop up. Select the file type (SQLite, JSON, or Excel), enter the path of the file to export, and accept the dialog.

Note: Export operations include all uncommitted changes.

11.8.4 Accessing/using exported files

Whenever you successfully export a file, a button with the file name is created in the *Exports* bar at the bottom of the form. To open the file in your registered program, press that button. To open the containing folder, click on the arrow next to the file name and select **Open containing folder** from the popup menu.

11.9 Committing and rolling back

Note: Changes are not immediately saved to the database(s). They need to be committed separately.

To commit your changes, select **Session -> Commit** from the hamburger menu, enter a commit message and press **Commit**. Any changes made in the current session will be saved into the database.

To undo *all* changes since the last commit, select **Session -> Rollback** from the hamburger menu.

Tip: To undo/redo individual changes, use the **Undo** and **Redo** actions from the **Edit** menu.

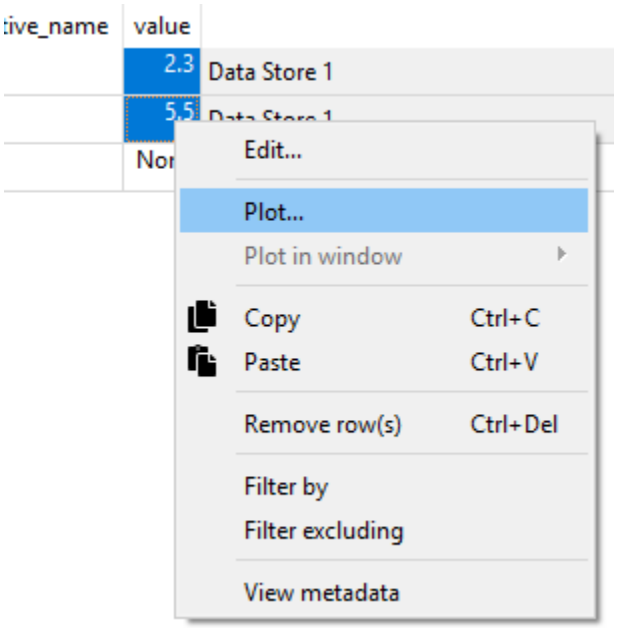
CHAPTER

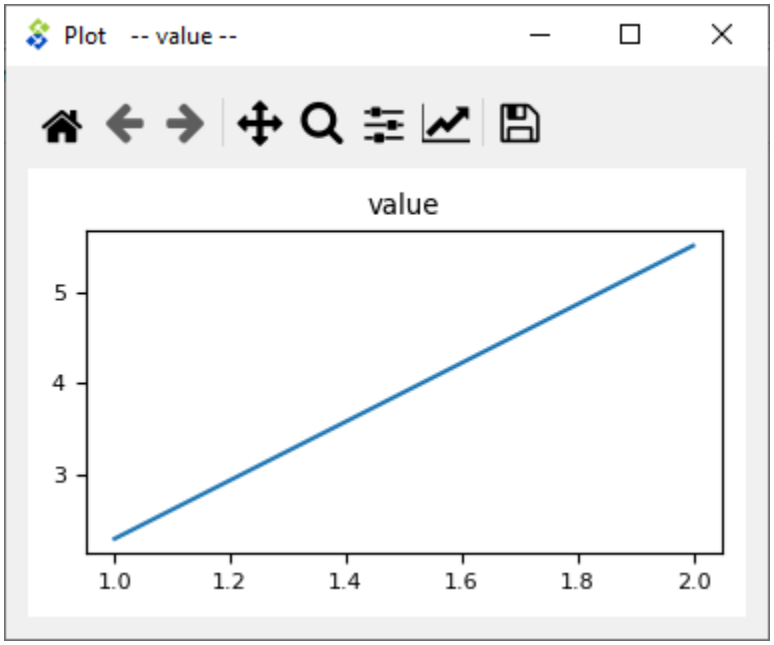
TWELVE

PLOTTING

Basic data visualization is available in the Spine database editors. Currently, it is possible to plot scalar values as well as time series, arrays and one dimensional maps with some limitations.

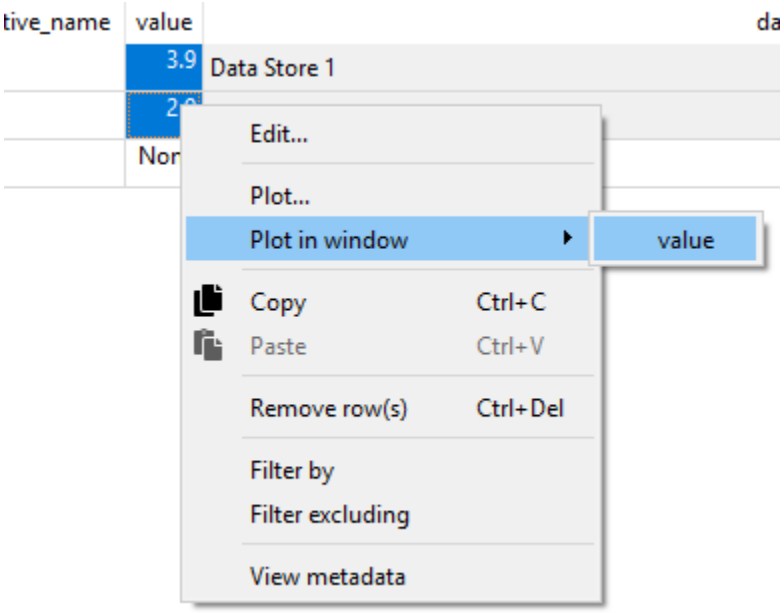
To plot a column, select the values from a table and then *Plot* from the **right click** popup menu.

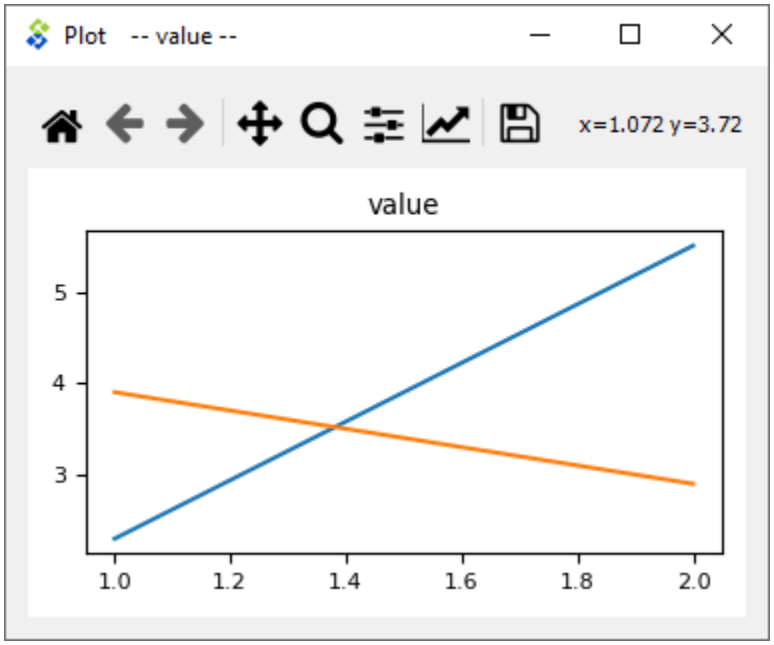




Selecting data in multiple columns plots the selection in a single window.

To add a plot to an existing window select the target plot window from the *Plot in window* submenu.

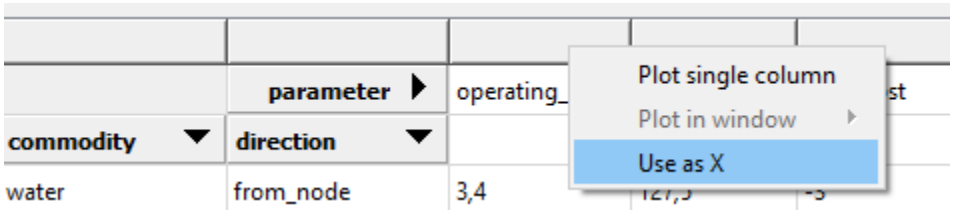




12.1 X column in pivot table

It is possible to plot a column of scalar values against a designated X column in the pivot table.

To set a column as the X column **right click** the top empty area above the column header and select *Use as X* from the popup menu. *(X)* in the topmost cell indicates that the column is designated as the X axis.



When selecting and plotting other columns in the same table the data will be plotted against the values in the X column instead of row numbers.

CHAPTER

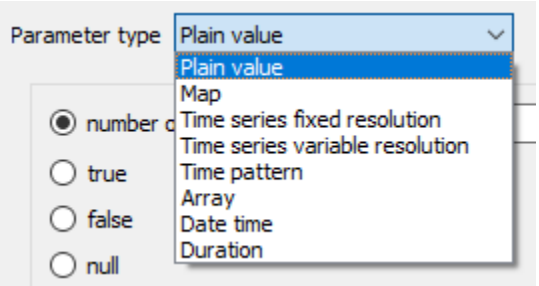
THIRTEEN

PARAMETER VALUE EDITOR

Parameter value editor is used to edit object and relationship parameter values such as time series, time patterns or durations. It can also convert between different value types, e.g. from a time series to a time pattern.

The editor is available from a **right click** popup menu or by **double clicking** a parameter value in one of the Spine database editors.

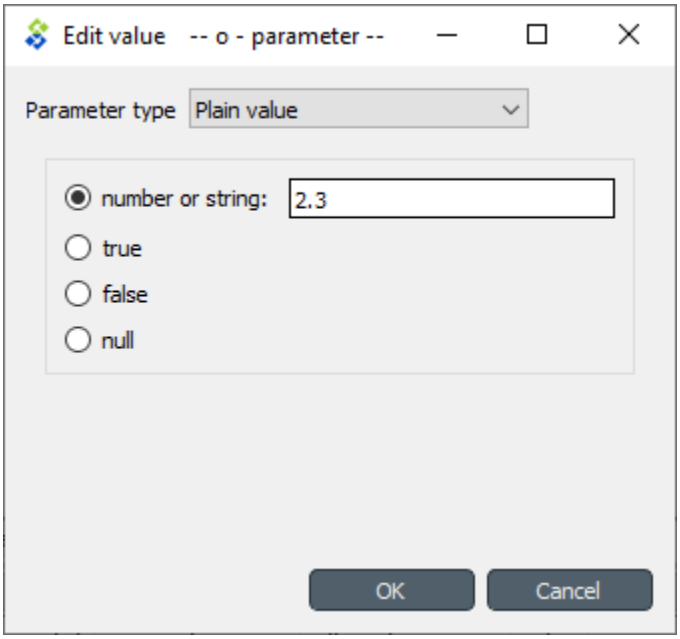
13.1 Choosing value type



The combo box at the top of the editor window allows changing the type of the current value.

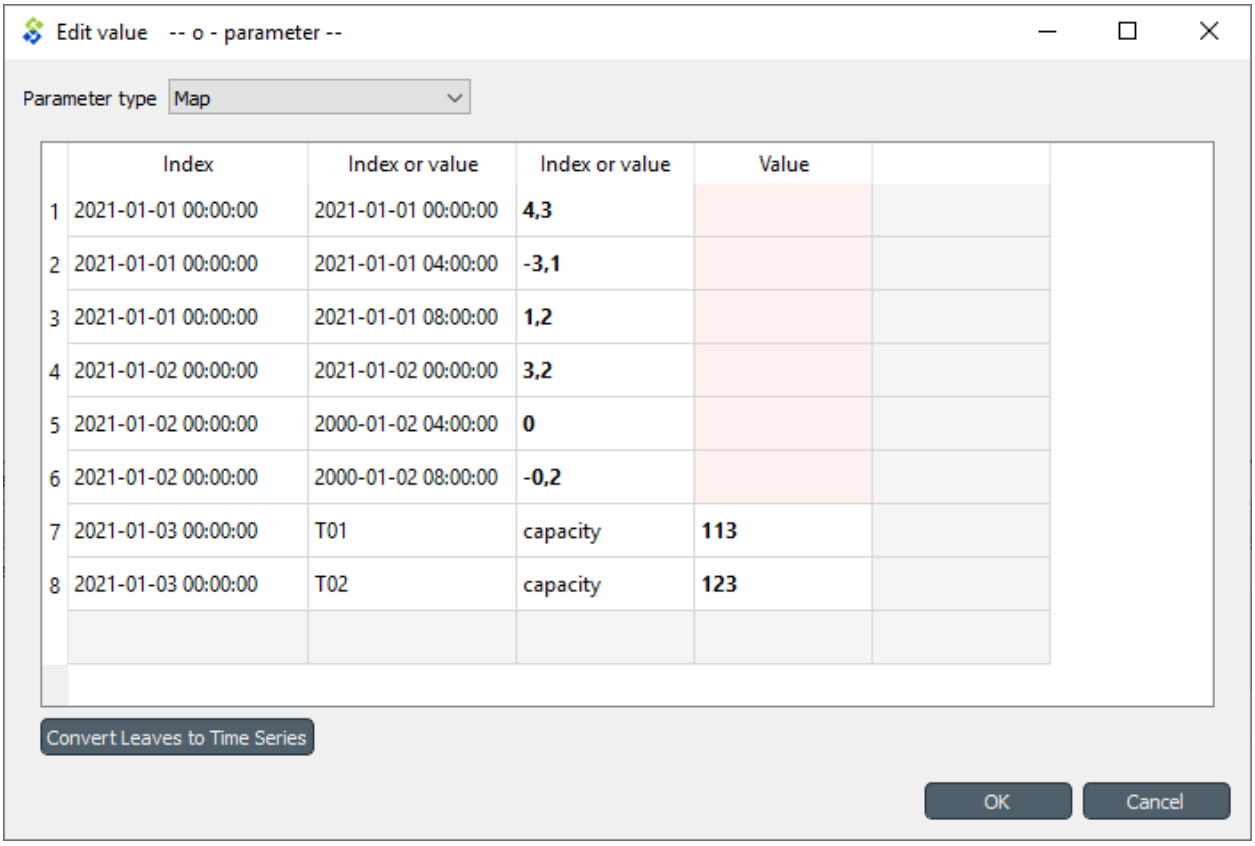
13.2 Plain values

The simplest parameter values are of the *Plain value* type. The editor window lets you to write a number or string directly to the input field or set it to true, false or null as needed.



13.3 Maps

Maps are versatile nested data structures designed to contain complex data including one and multi dimensional indexed arrays. In Parameter value editor a map is shown as a table where the last non-empty cell on each row contains the value while the preceding cells contain the value's indexes.

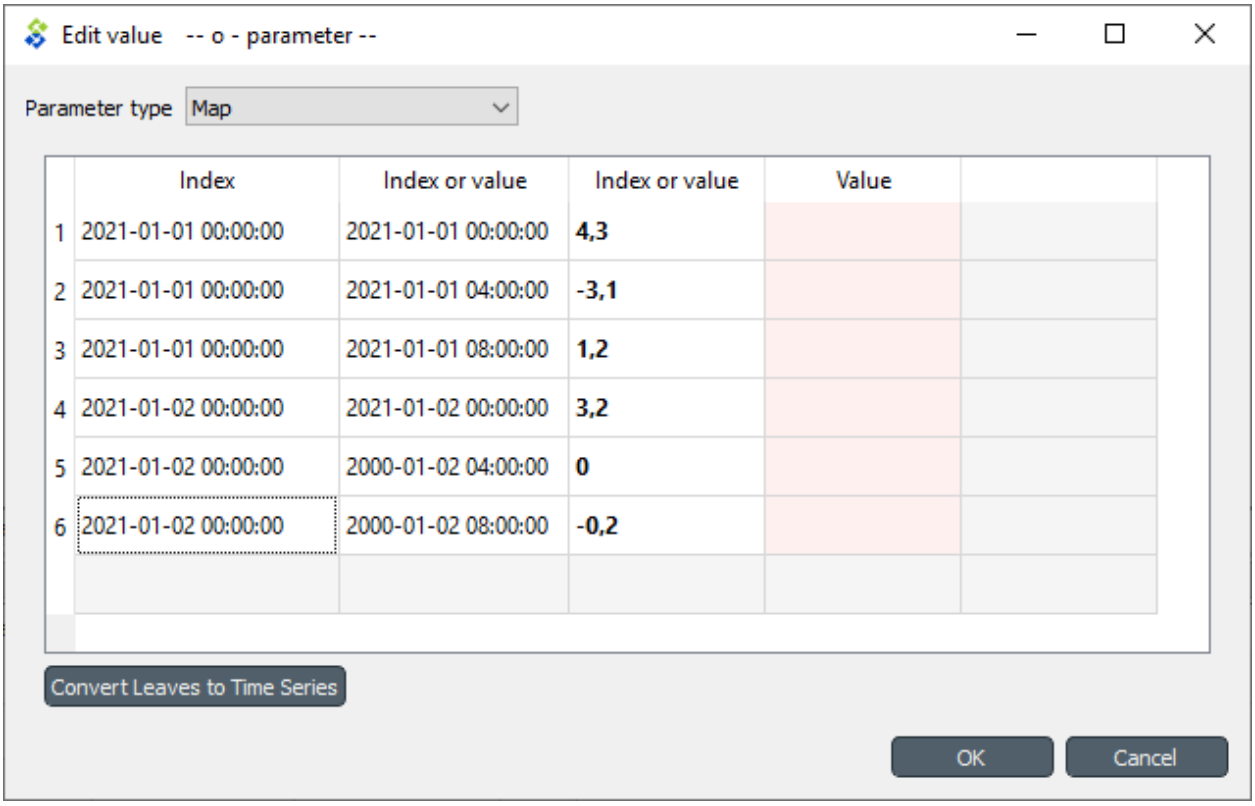


The extra gray column on the right allows expanding the map with a new dimension. You can append a value to the map by editing the bottom gray row. The reddish cells are merely a guide for the eye to indicate that the map has different nesting depths.

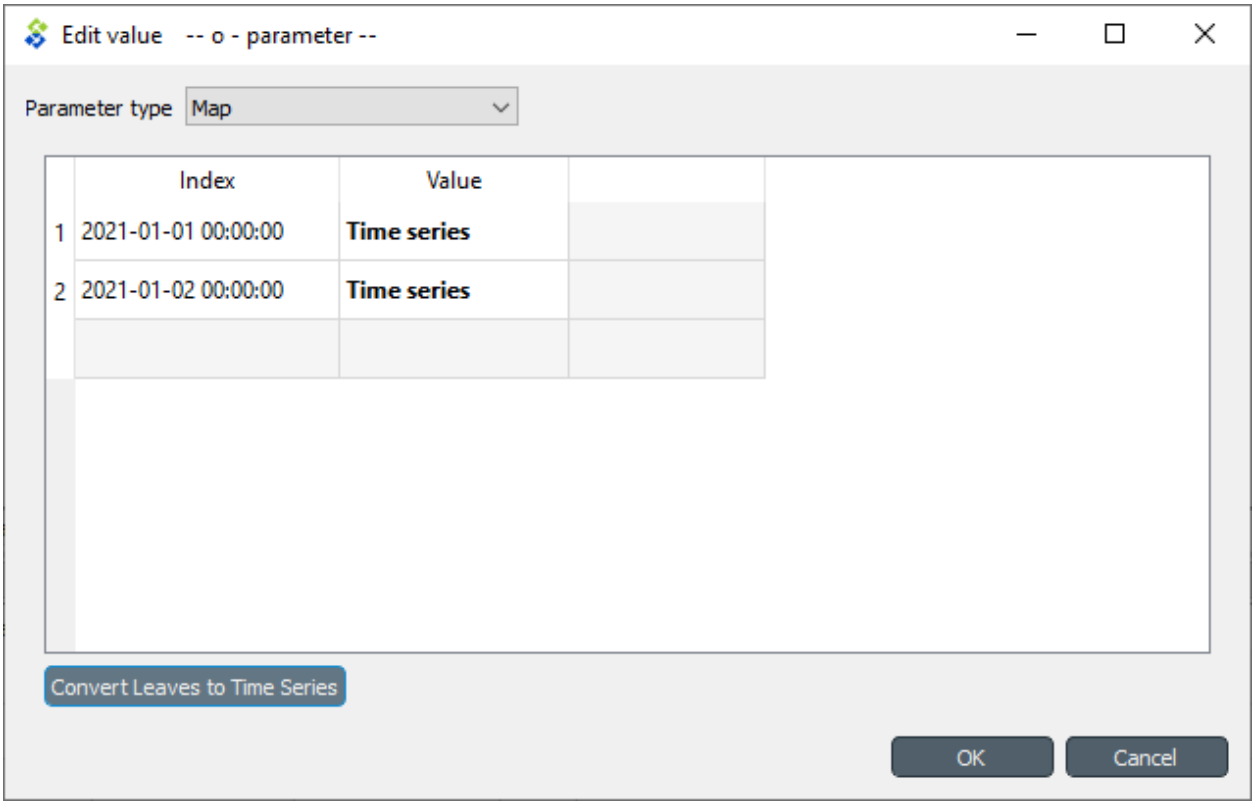
A **Right click** popup menu gives options to open a value editor for individual cells, to add/insert/remove rows or columns (effectively changing map's dimensions), or to trim empty columns from the right hand side.

Copying and pasting data between cells and external programs works using the usual **Ctrl-C** and **Ctrl-V** keyboard shortcuts.

Convert leaves to time series 'compacts' the map by converting the last dimension into time series. This works only if the last dimension's type is datetime. For example the following map contains two time dimensions. Since the indexes are datetimes, the 'inner' dimension can be converted to time series.

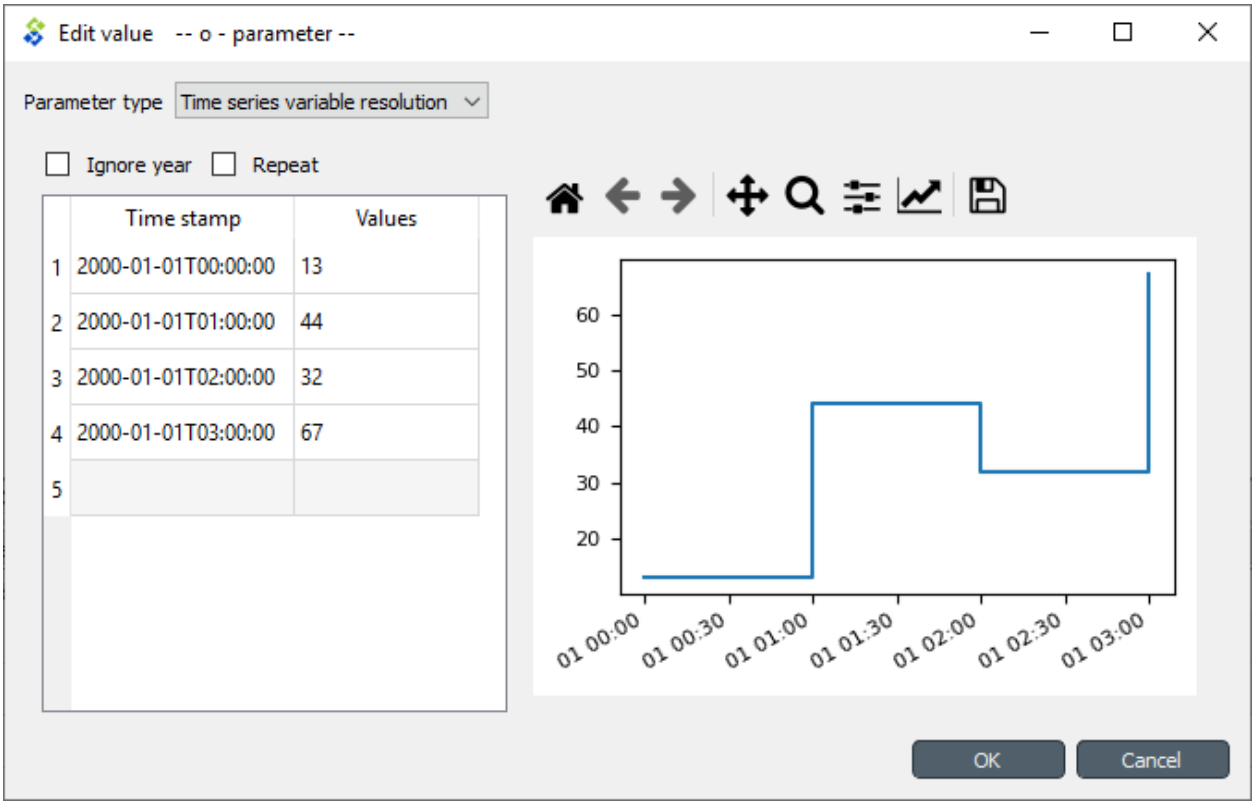


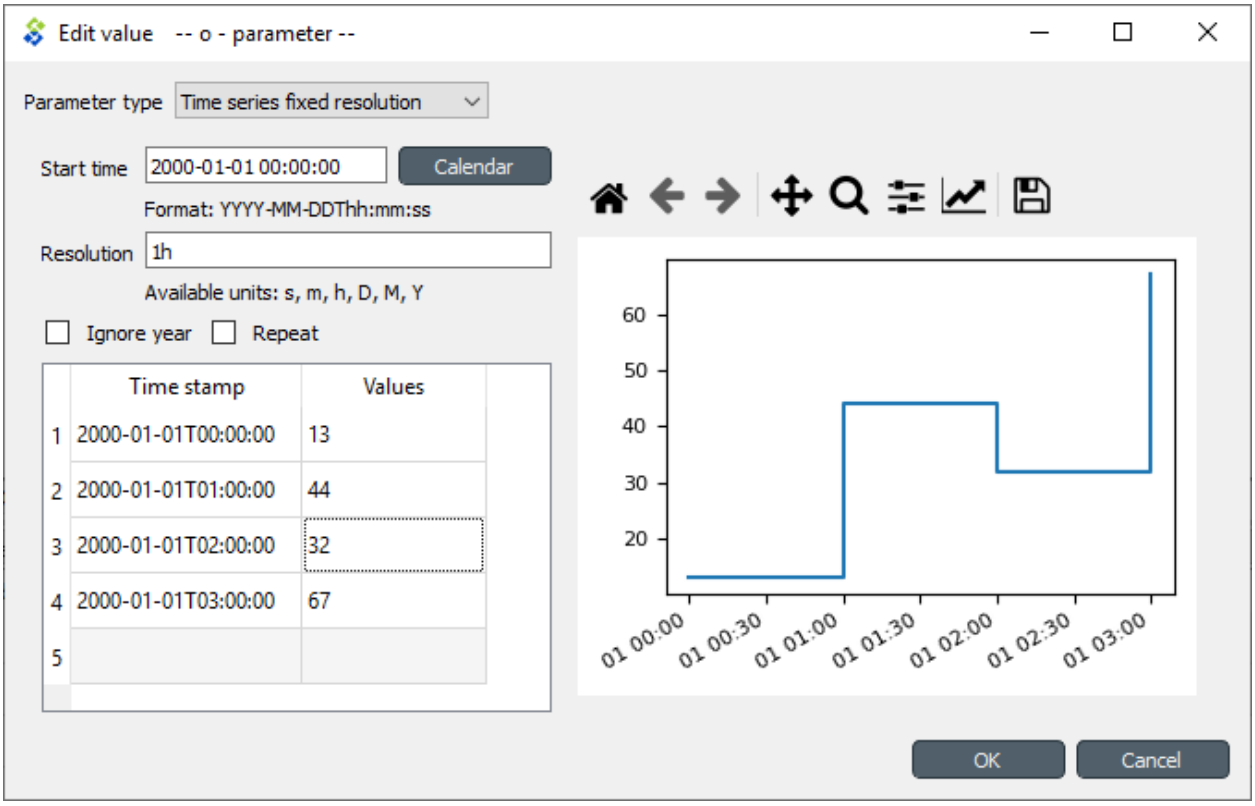
After clicking **Convert leaves to time series** the map looks like this:



13.4 Time series

There are two types of time series: *variable* and *fixed resolution*. Variable resolution means that the time stamps can be arbitrary while in fixed resolution series the time steps between consecutive stamps are fixed.





The editor window is split into two in both cases. The left side holds all the options and a table with all the data while the right side shows a plot of the series. The plot is not editable and is for visualization purposes only.

In the table rows can be added or removed from a popup menu available by a **right click**. Editing the last gray row appends a new value to the series. Data can be copied and pasted by **Ctrl-C** and **Ctrl-V**. Copying from/to an external spreadsheet program is supported.

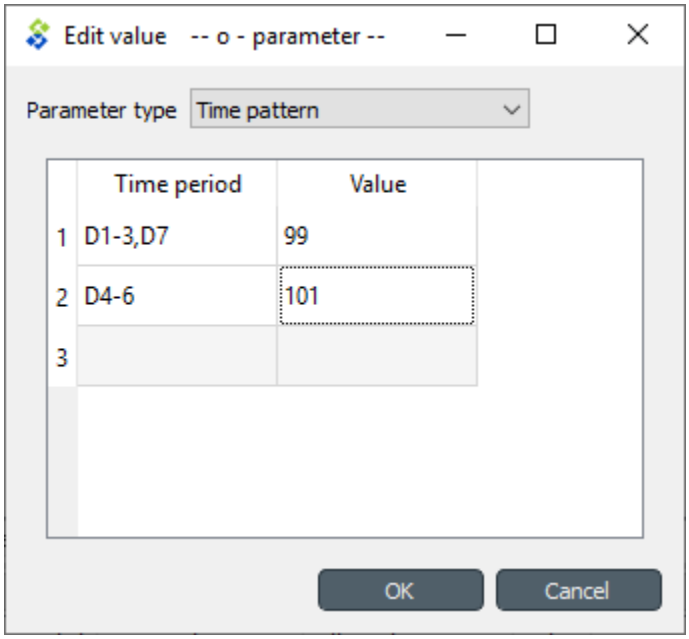
The time steps of a fixed resolution series are edited by the *Start time* and *Resolution* fields. The format for the start time is ISO8601. The *Resolution* field takes a single time step or a comma separated list of steps. If a list of resolution steps is provided then the steps are repeated so as to fit the data in the table.

The *Ignore year* option available for both variable and fixed resolution time series allows the time series to be used independent of the year. Only the month, day and time information is used by the model.

The *Repeat* option means that the time series is cycled, i.e. it starts from the beginning once the time steps run out.

13.5 Time patterns

The time pattern editor holds a single table which shows the *time period* on the right column and the corresponding values on the left. Inserting/removing rows and copy-pasting works as in the time series editor.

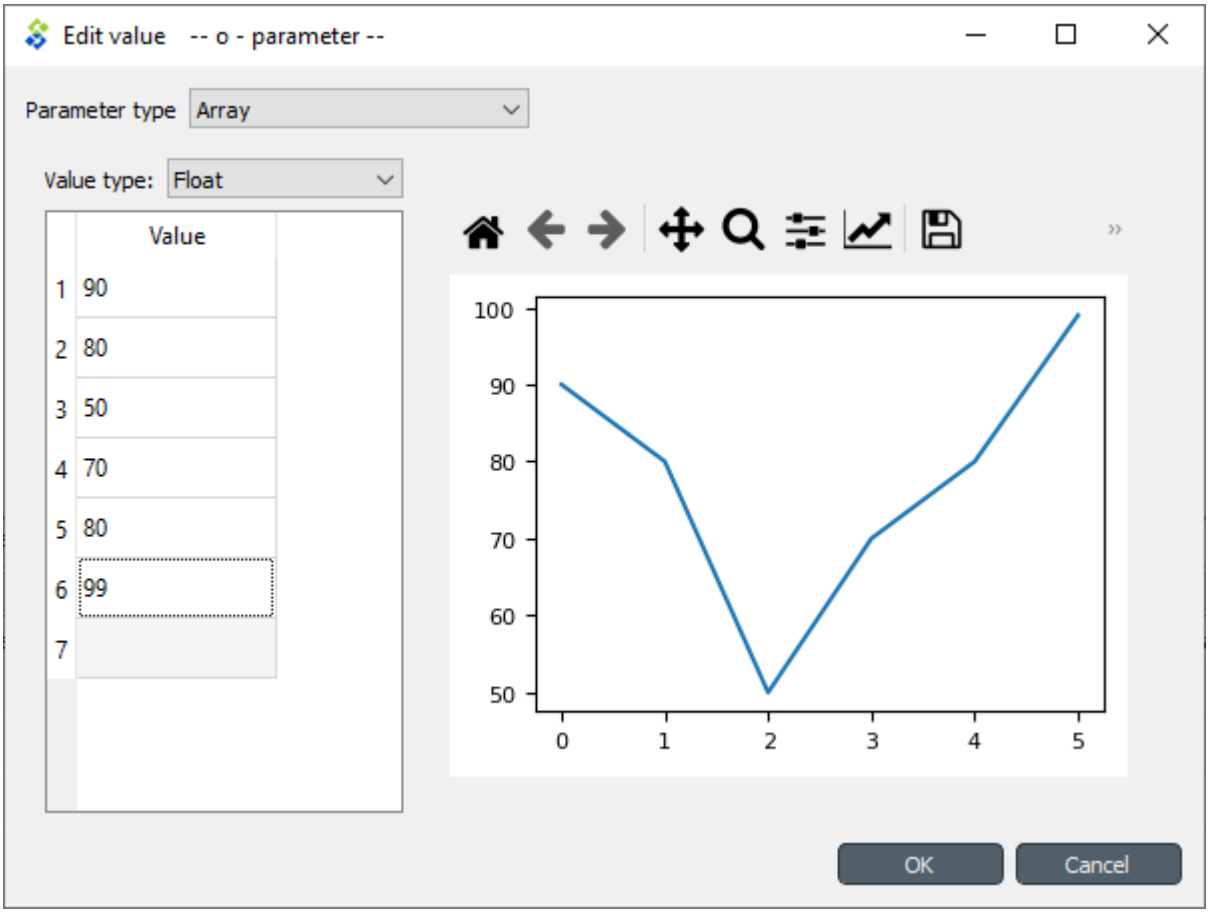


Time periods consist of the following elements:

- An *interval* of time in a given *time-unit*. The format is `Ua-b`, where `U` is either `Y` (for year), `M` (for month), `D` (for day), `WD` (for weekday), `h` (for hour), `m` (for minute), or `s` (for second); and `a` and `b` are two integers corresponding to the lower and upper bound, respectively.
- An *intersection* of intervals. The format is `s1;s2;...`, where `s1`, `s2`, ..., are intervals as described above.
- A *union of ranges*. The format is `r1,r2,...`, where `r1`, `r2`, ..., are either intervals or intersections of intervals as described above.

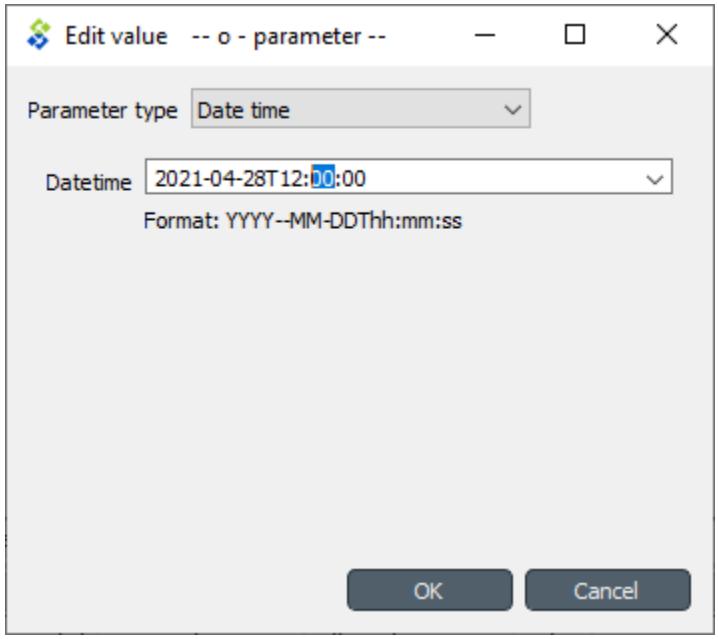
13.6 Arrays

Arrays are lists of values of a single type. Their editor is split into two: the left side holds the actual array while the right side contains a plot of the array values versus the values' positions within the array. Note that not all value types can be plotted. The type can be selected from the *Value type* combobox. Inserting/removing rows and copy-pasting works as in the time series editor.



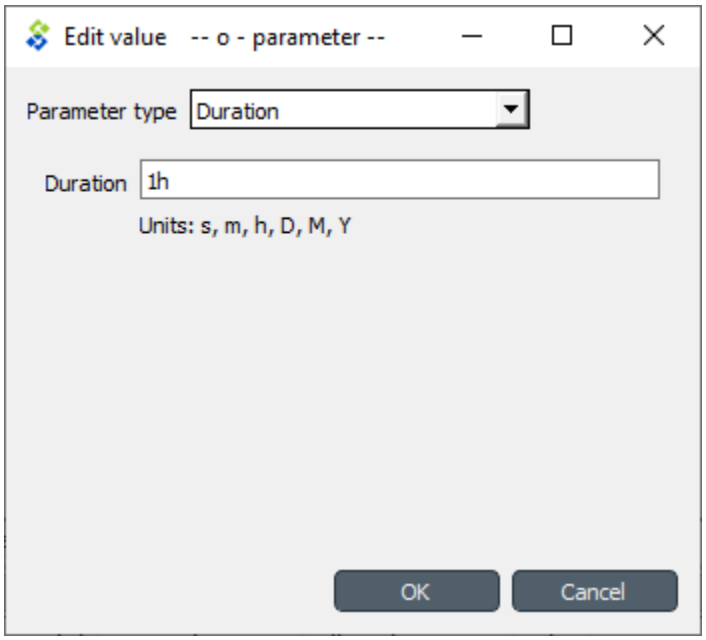
13.7 Datetimes

The datetime value should be entered in ISO8601 format. Clicking small arrow on the input field pops up a calendar that can be used to select a date.



13.8 Durations

A single value or a comma separated list of time durations can be entered to the *Duration* field.



CHAPTER

FOURTEEN

IMPORTING AND EXPORTING DATA

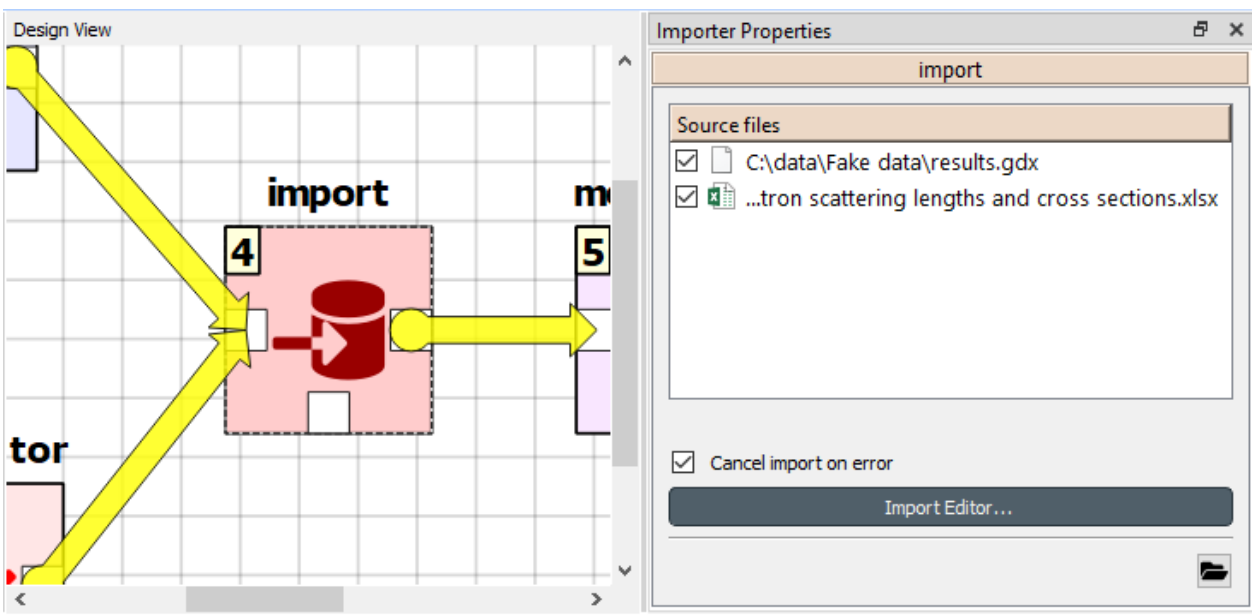
This section explains the different ways of importing and exporting data to and from a Spine database.

14.1 Importing data with Importer

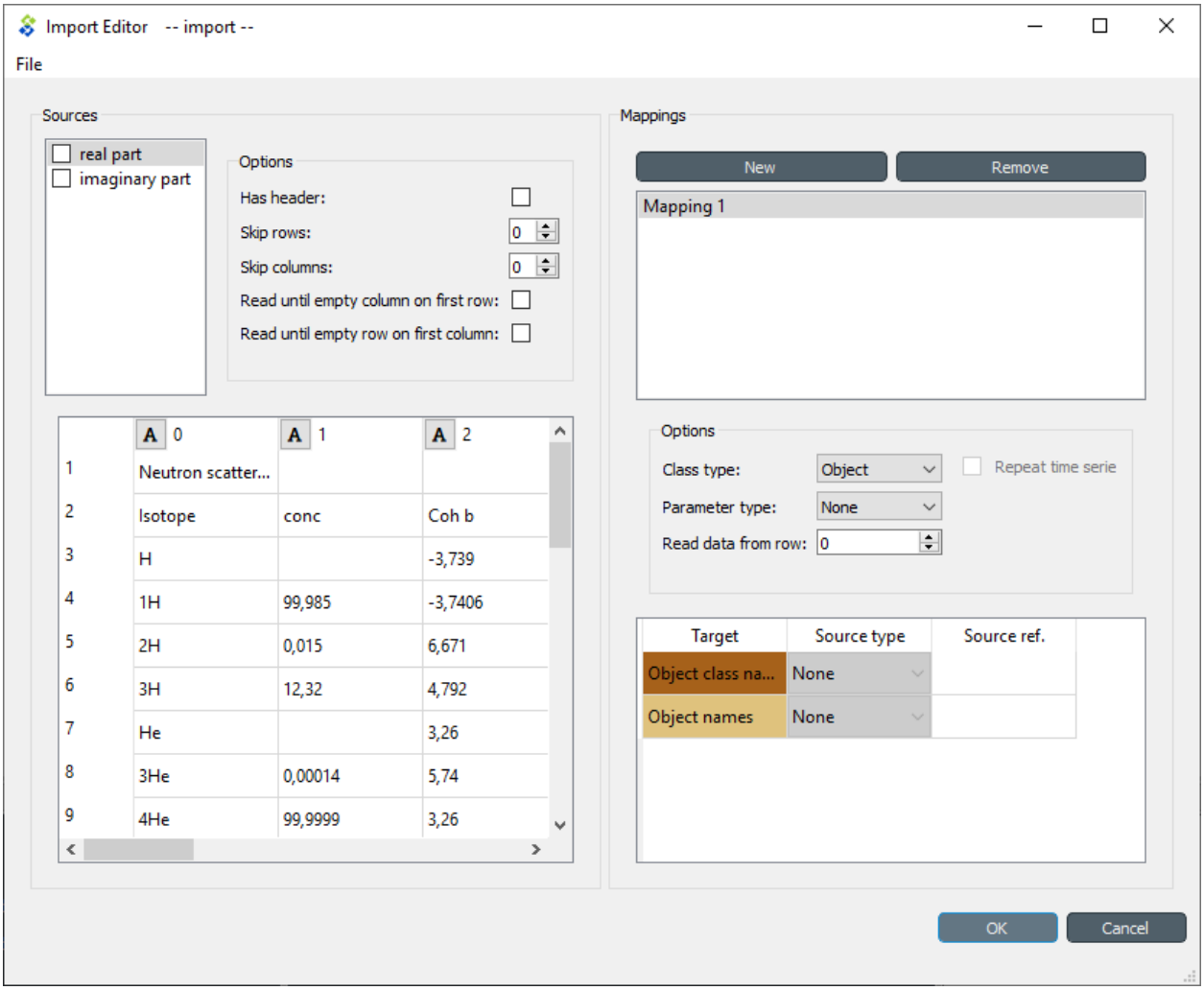
Data importing is handled by the Importer project item which can import tabulated and to some degree tree-structured data into a Spine database from various formats. The same functionality is also available in **Spine database editor** from **File->Import** but using an Importer item is preferred because then the process is documented and repeatable.

Tip: A Tool item can also be connected to Importer to import tool's output files to a database.

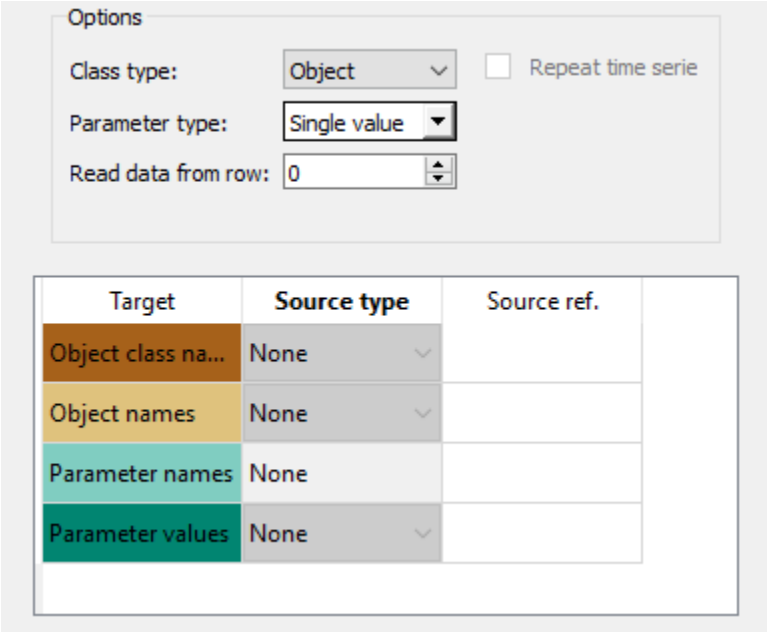
The heart of Importer is the **Import Editor** window in which the mappings from source data to Spine database entities are set up. The editor window can be accessed by the **Import Editor…** button in Importer's Properties dock. Note, that you have to select one of the files in the **Source files** list before clicking the button.



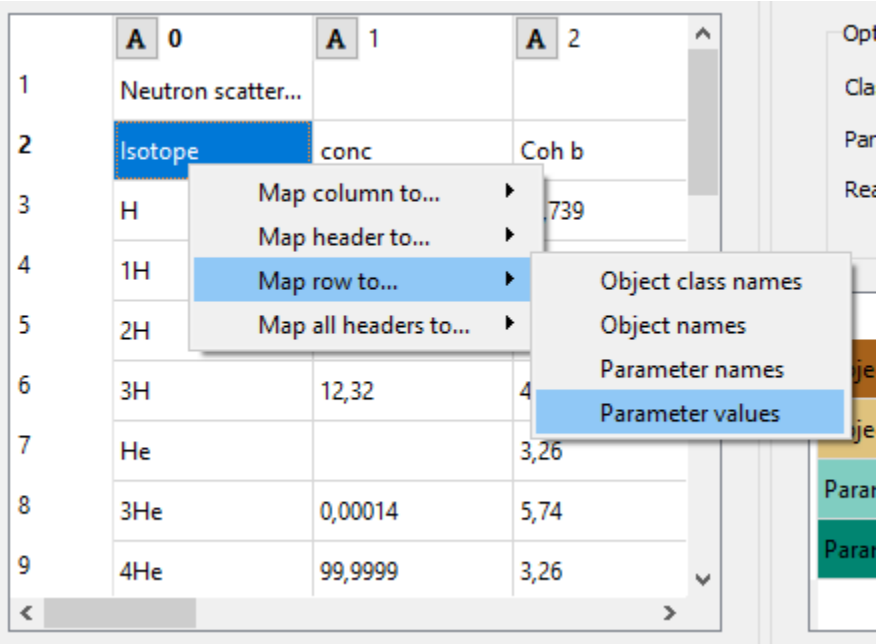
The **Import Editor** windows is divided into two parts: **Sources** shows all the 'sheets' contained in the file, some options for reading the file correctly, and a preview table to visualize and configure how the data on the selected sheet would be mapped. **Mappings**, on the other hand, shows the actual importing settings, the mappings from the input data to database entities.



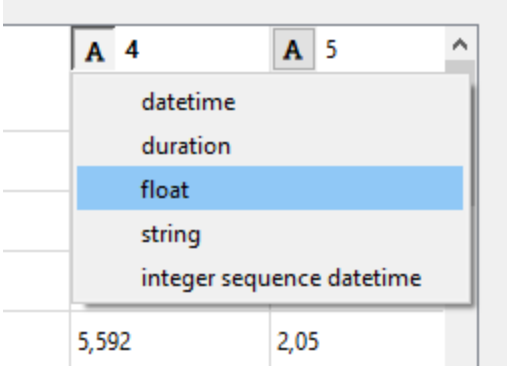
The options in the Mappings part declare if the currently selected sheet will be imported as an object or relationship and what type of parameters, if any, the sheet contains. The table can be used to configure how the input data is interpreted: which row or column contains the entity class names, parameter values, time stamps and so on.



It might be helpful to fill in the mapping options using the preview table in the Sources part. Right clicking on the table cells shows a popup menu that lets one to configure how the rows and colunms are read upon importing.



An important aspect of data import is whether each item in the input data should be read as a string, a number, a time stamp, or something else. By default all input data is read as strings. However, more often than not things like parameter values are actually numbers. It is possible to control what type of data each column (and, sometimes, each row) contains from the preview table. Clicking the data type indicator button on column headers pops up a menu with a selection of available data types. Right clicking the column header also gives the opportunity to change the data type of all columns at once.



14.2 Exporting data with Exporter

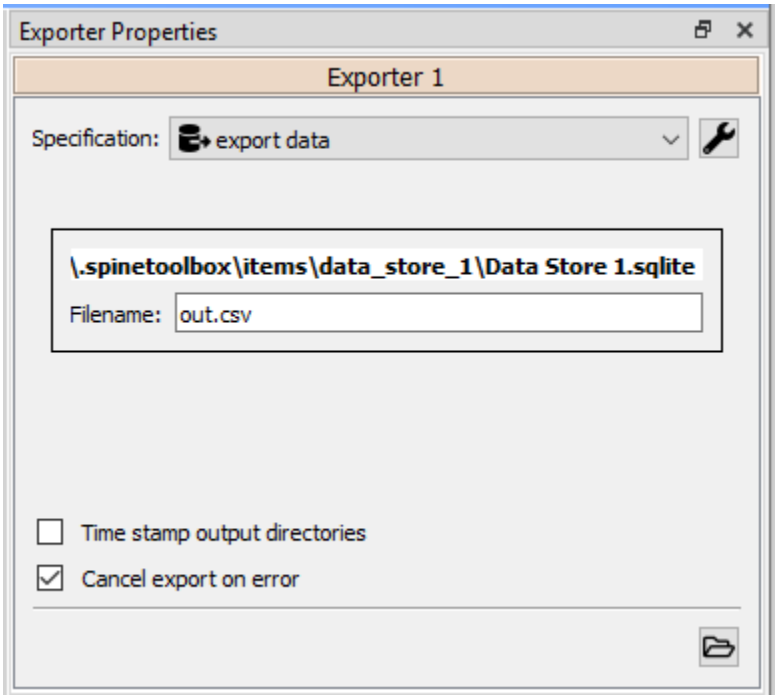
Exporter writes database data into regular files that can be used by Tools and external software that do not read the Spine database format. Various tabulated file formats are supported some of which require specific export settings; see below for more details.

At its heart Exporter maps database items such as entity class or entity names to an output table. Each item has a user given output **position** on the table, for example a column number. By default data is mapped to columns but it is also possible to create pivot tables.

Exporter saves its settings or export **mappings** as a specification that can be reused by other exporters or even other projects. The specification can be edited in *Exporter specification editor* which is accessible by the button in the item's Properties dock or by double clicking exporter's icon on the Design view. A specification that is not associated with any specific Exporter project item can be created and edited from the Main toolbar.

14.2.1 Properties dock

Exporter's Properties dock controls project item specific settings that are not part of the item's specification.



Specification used by the active Exporter item can be selected from the *Specification* combobox. The button opens *Exporter specification editor* where it is possible to edit the specification.

Databases available for export from connected project items such as Data stores are listed in separate boxes below the Specification combobox. An output filename is required for each database.

Checking the *Time stamp output directories* box adds a time stamp to the item's output directories preventing output files from being overwritten. This may be useful for debugging purposes.

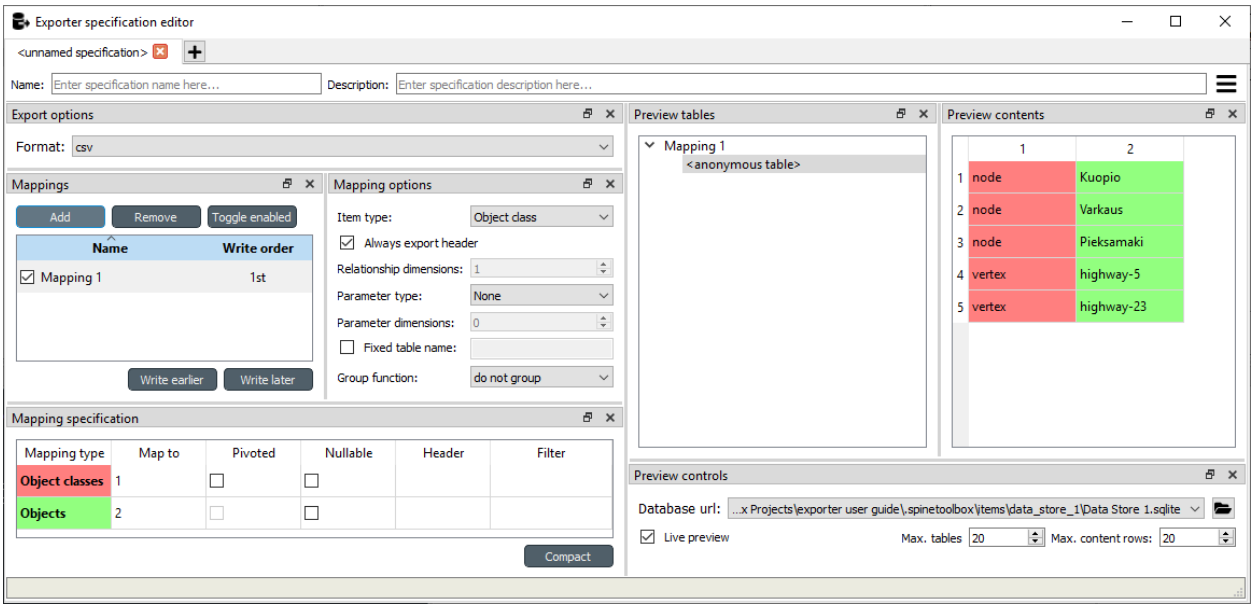
The *Cancel export on error* checkbox controls whether execution bails out on errors that may be otherwise non-fatal.

Exporter's data directory can be opened in system's file browser by the button. The output files are written in data directory's `output` subdirectory.

14.2.2 Exporter specification editor

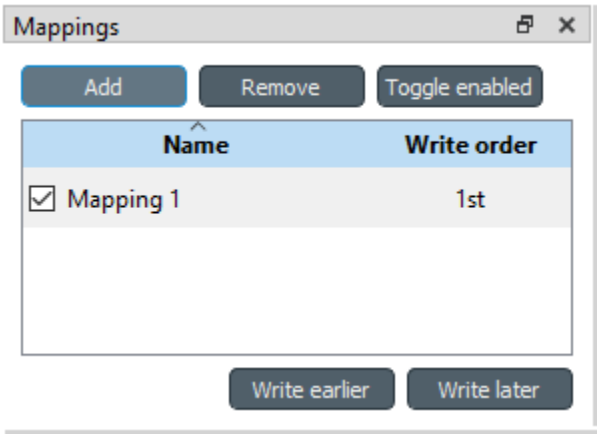
Specification editor is used to create **mappings** that define how data is exported to the output file. Mappings define one or more tables and their contents but are otherwise output format agnostic. Some output formats, e.g. SQL and gdx, interpret the tables in specific ways, however. Other formats which inherently cannot write multiple tables into a single file, such as csv, may end up exporting multiple files. See the sections below for format specific intricacies.

When opened for the first time Specification editor looks like in the figure below. The window is tabbed allowing multiple specifications to be edited at the same time. Each tab consists of dock widgets which can be reorganized to suit the user's needs. The 'hamburger' menu on the top right corner gives access to some important actions such as *Save* and *Close*. *Undo* and *redo* can be found from the menu as well.



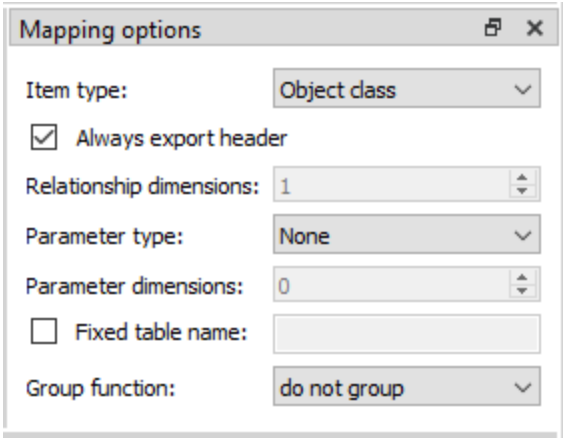
The only requirement for a specification is a name. This can be given on the *Name* field field on the top bar. The *Description* field allows for an additional explanatory text.

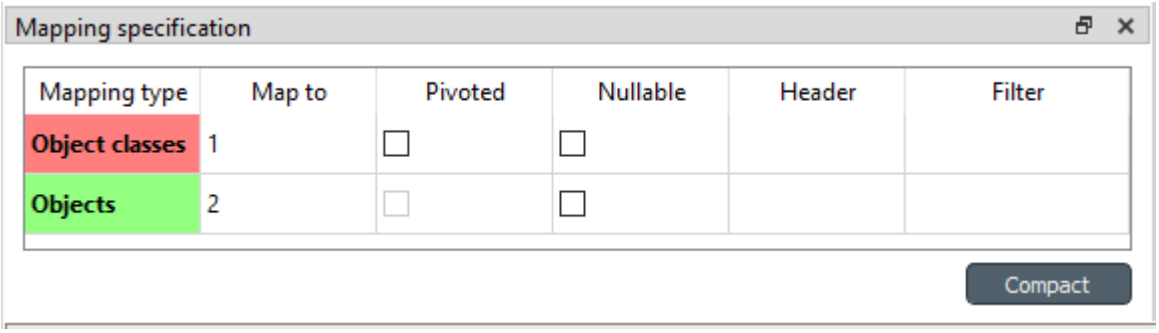
The current output format can be changed by the *Format* combobox on *Export options* dock.



Specification's mappings are listed in the *Mappings* dock shown above. The *Add* button adds a new mapping while the *Remove* button removes selected mappings. Mappings can be renamed by double clicking their names on the list. The checkbox in front of mapping's name shows if the mapping is currently enabled. Use the *Toggle enabled* button to toggle the enabled state of all mappings at once.

The tables defined by the mappings are written in the order shown on the mapping list's *Write order* column. This may be important if the tables need to be in certain order in the output file or when multiple mappings output to a single table. Mappings can be sorted by their write order by clicking the header of the *Write order* column. The *Write earlier* and *Write later* buttons move the currently selected mapping up and down the list.





Currently selected mapping is edited using the controls in *Mapping options* and *Mapping specification* docks. The *Mapping options* dock contains controls that apply to the mapping as a whole, e.g. what data the output tables contain. *Mapping specification*, on the other hand, contains a table which defines the structure of the mapping's output tables.

What database items the mapping outputs is chosen using the *Item type* combobox in *Mapping options* dock. For instance, the *Object classes* option outputs object classes, objects and, optionally, object parameters and related items while the *Relationship classes* option outputs relationship classes and relationships. Checking the *Always export header* checkbox outputs a table that has fixed headers even if the table is otherwise empty. If *Item type* is Relationship class, the *Relationship dimensions* spinbox can be used to specify the maximum number of relationships' dimensions that the mapping is able to handle. Parameters can be outputted by choosing their value type using the *Parameter type* combobox. The *Value* choice adds rows to *Mapping specification* for parameter values associated with individual entities while *Default value* allows outputting parameters' default values. The maximum number of value dimensions in case of indexed values (time series, maps, time patterns, arrays) the mapping can handle is controlled by the *Parameter dimensions* spinbox. The *Fixed table name* checkbox enables giving a user defined table name to the mapping's output table. In case the mapping is pivoted and *Mapping specification* contains items that are *hidden*, it is possible that a number of data elements end up in the same output table cell. The *Group function* combobox offers some basic functions to aggregate such data into the cells.

The contents of the table on the *Mapping specification* dock depends on choices on *Mapping options*, e.g. the item type, parameter type or dimensions. Each row corresponds to an item in the database: object class names, object names, parameter values etc. The item's name is given in the *Mapping type* column. The colors help to identify the corresponding elements in the preview. The *Map to* column defines the **position** of the item, that is, where the item is written or otherwise used when the output tables are generated. By default, a plain integral number in this column means that the item is written to that column in the output table. From the other choices, *hidden* means that the item will not show on the output. *Table name*, on the other hand, uses the item as output table names. For example, outputting object classes as table names will generate one new table for every object class in the database, each named after the class. Each table in turn will contain the parameters and objects of the table's object class. If multiple mappings generate a table with a common name then each mapping appends to the same table in the order specified by the *Write order* column on *Mappings* dock. The *column header* position makes the item a column header for a **buddy item**. Buddy items have some kind of logical relationship with their column header, for instance the buddy of an object class

is its objects; setting the object class to *column header* will write the name of the class as the objects' column header.

Note: Currently, buddies are fixed and defined only for a small set database items. Therefore, *column header* will not always produce sensible results.

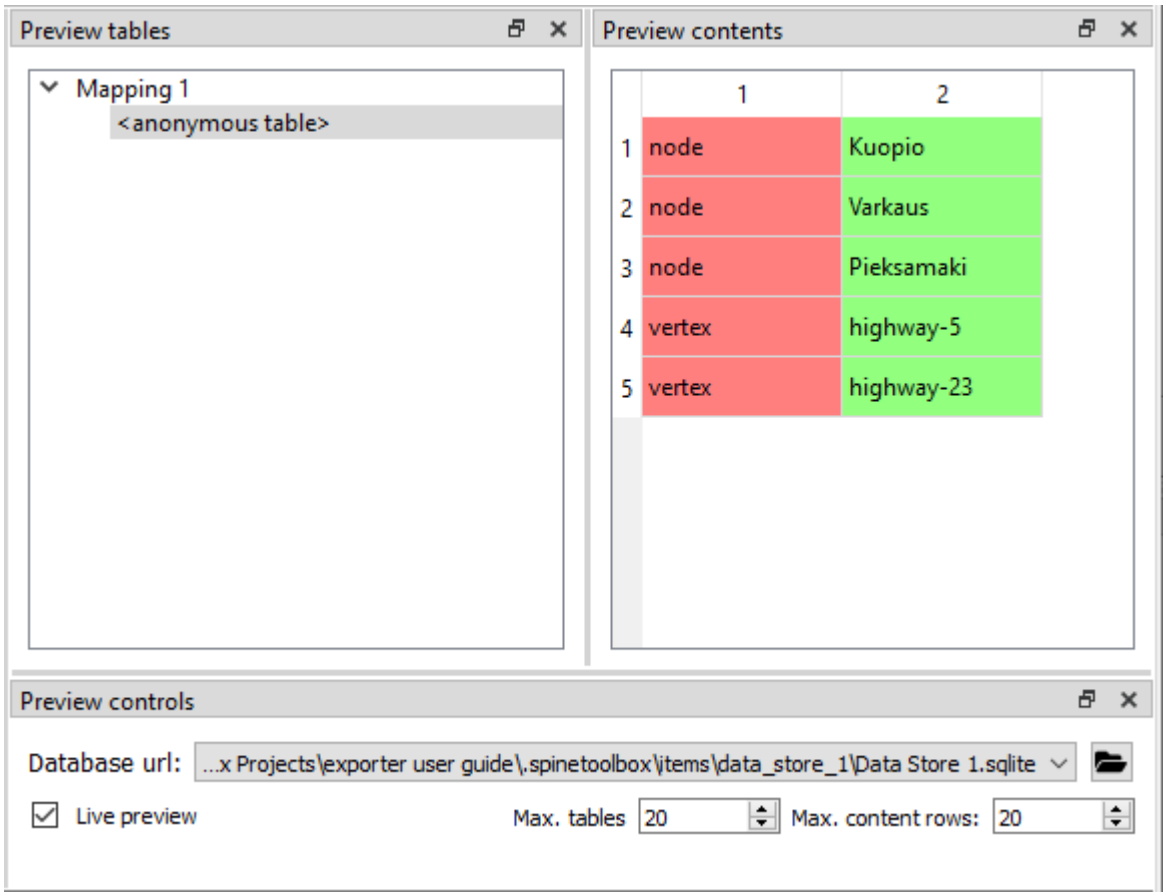
Changing the column and pivot header row positions leaves sometimes gaps in the output table. If such gaps are not desirable the *Compact* button reorders the positions by removing the gaps. This may be useful when the output format requires such gapless tables.

The checkboxes in *Pivoted* column on the *Mapping specification* dock toggle the mapping into pivoted mode. One or more items on the table can be set as pivoted. They then act as a pivot header for the data item which is the last non-hidden item on the list. Once checked as pivoted, an item's position column defines a pivot header row instead of output column.

By default a row ends up in the output table only when all mapping items yield some data. For example, when exporting object classes and objects, only classes that have objects get written to output. However, sometimes it is useful to export 'empty' object classes as well. For this purpose a mapping can be set as **nullable** in the *Nullable* column. Continuing the example, checking the *Nullable* checkbox for *Objects* would produce an output table with all object classes including ones without objects. The position where objects would normally be outputted are left empty for those classes.

Besides the *column header* position it is possible give fixed column headers to items using the *Header* column in *Mapping specification* dock. Note that checking the *Always export header* option in the *Mapping options* dock outputs the fixed headers even if there is no other data in a table.

The *Mapping specification* dock's *Filter* column provides refined control on which database items the mapping outputs. The column uses regular expressions to filter what gets outputted. See Basic regular expression for filtering.



A preview of what will be written to the output is available in the preview dock widgets. A database connection is needed to generate the preview. The *Preview controls* dock provides widgets to choose an existing database or to load one from a file. Once a database is available and the preview is enabled the mappings and the tables they would output are listed on the *Preview tables* dock. Selecting a table from the list shows the table's contents on the *Preview contents* dock. The colors on the table correspond to the colors in *Mapping specification* dock.

14.2.3 Basic regular expressions for filtering

The *Filter* field in *Mapping specification* accepts regular expressions to filter what data gets outputted by that mapping item. Below are examples on how to create some basic filters.

Single item

Writing the item's name to the field filters out all other items. For example, to output the object class called 'node' only, write `node` to the *Filter* field.

OR operator

The vertical bar `|` serves as the OR operator. `node|unit` as a filter for object classes would output classes named 'node' and 'unit'.

Excluding an item

While perhaps not the most suitable task for regular expressions it is still possible to 'negate' a filter. *^(?!node).* would exclude all items names of which start with 'node'.

14.2.4 Csv and multiple tables

Csv files are flat text files and therefore do not directly support multiple tables. Instead, multiple tables are handled as separate output files.

Only mappings that output an **anonymous table** actually write to the file specified on the Exporter's properties dock. Named tables get written to files named after the table plus the `.csv` extension. For example, a table named `node` would result in a file called *node.csv*.

14.2.5 SQL export

Note: Currently only sqlite is supported.

The SQL backend writes the tables to the target database in a relatively straightforward way:

- Tables are named after the table name provided by the mappings. **Anonymous tables** are not supported.
- The first row of each table is used as column names in the database. Thus, each column in a mapping should have a fixed header or a header produced by an item set to *column header* position.
- Column data types are sniffed from the second row. Empty values or a missing row result in string type.
- There must be an item assigned to each column. Empty columns confuse the SQL backend.
- Pivot tables do not generally make sense with the SQL backend unless the resulting table somehow follows the above rules.

14.2.6 GAMS gdx export

Note: You need to have GAMS installed to use this functionality. However, you do not need to own a GAMS license as the demo version works just as well.

Note: The bitness (32 or 64bit) of GAMS must match the bitness of the Python interpreter.

The gdx backend turns the output tables to GAMS sets, parameters and scalars following the rules below:

- Table names correspond the names of sets, parameters and scalars. Thus, **anonymous tables** are not supported.
- There must be an item assigned to each column. Empty columns confuse the gdx backend.
- Pivot tables do not generally make sense with the gdx backend unless the resulting table somehow follows the rules listed here.

Sets:

- Everything that is not identified as parameter or scalar is considered a GAMS set.
- Each column corresponds to a dimension.
- The first row is used to name the dimension's domain. Thus, each column in a mapping should have a fixed header or a header produced by an item set to *column header* position. Note that * is a valid fixed header and means that the dimension has no specific domain.

Parameters:

- A table that contains no header in the last (rightmost) column is considered a GAMS parameter.
- The last column should contain the parameter's values while the other columns contain the values' dimension.
- Dimensions' domains are taken from the header row, see **Sets** above. Note, that the value column must not have a header.

Scalars:

- A table that contains a numerical value in the top left cell is considered a GAMS scalar. Everything else (except the table name) is ignored.
- The data in the top left cell is the scalar's value.

14.3 Exporting to GAMS with GdxExporter

Note: GdxExporter is pending for removal and its use in new projects is discouraged. Use Exporter instead.

Note: You need to have GAMS installed to use this functionality. However, you do not need to own a GAMS license as the demo version works just as well.

Note: The bitness (32 or 64bit) of GAMS has to match the bitness of the Python interpreter.

Databases can be exported to GAMS `.gdx` files by the *GdxExporter* project item. When a project is executed, *GdxExporter* writes its output files to its data folder and forwards file paths to project items downstream. If a *Tool* is to use such a file, remember to add the file as one of the *Tool specification*'s input files!

The mapping between entities in a Spine database and GAMS is as follows:

Database entity	GAMS entity
Object class	Universal set (or domain)
Object	Universal set member
Object parameter	Parameter
Relationship class	Subset of universal sets
Relationship	Subset member
Relationship parameter	Parameter

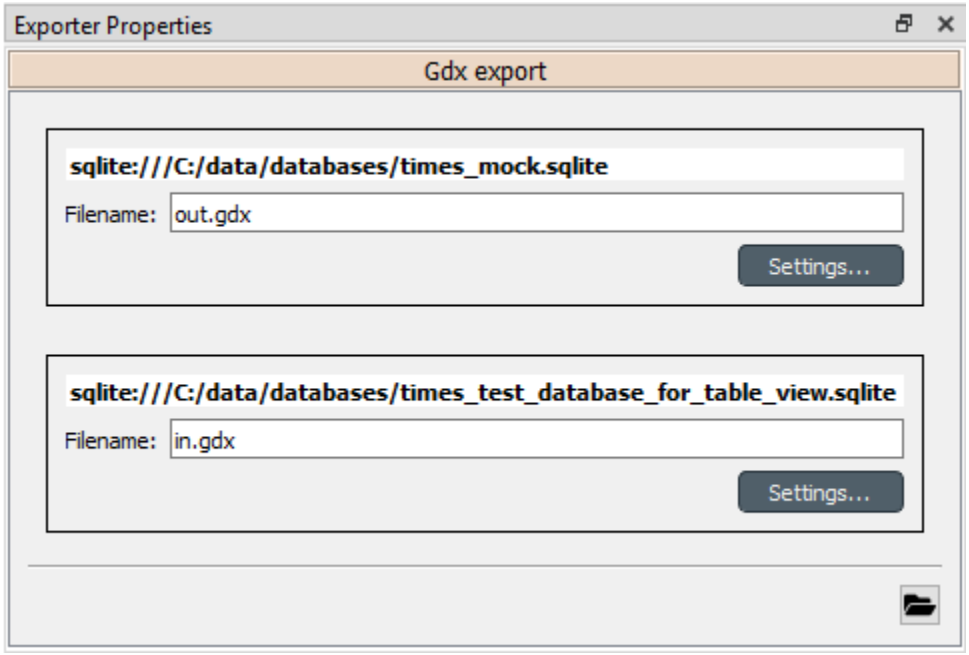
Note: Currently, it is not possible to use subsets (relationship classes) as dimensions for other subsets due to technical limitations. For example, if there is a domain **A(*)** and a subset **foo(A)**, a subset of **foo** has to be expressed as **bar(A)** instead of **bar(foo)**.

It is also possible to designate a single object class as a *Global parameter*. The parameters of the objects of that class will be exported as GAMS scalars.

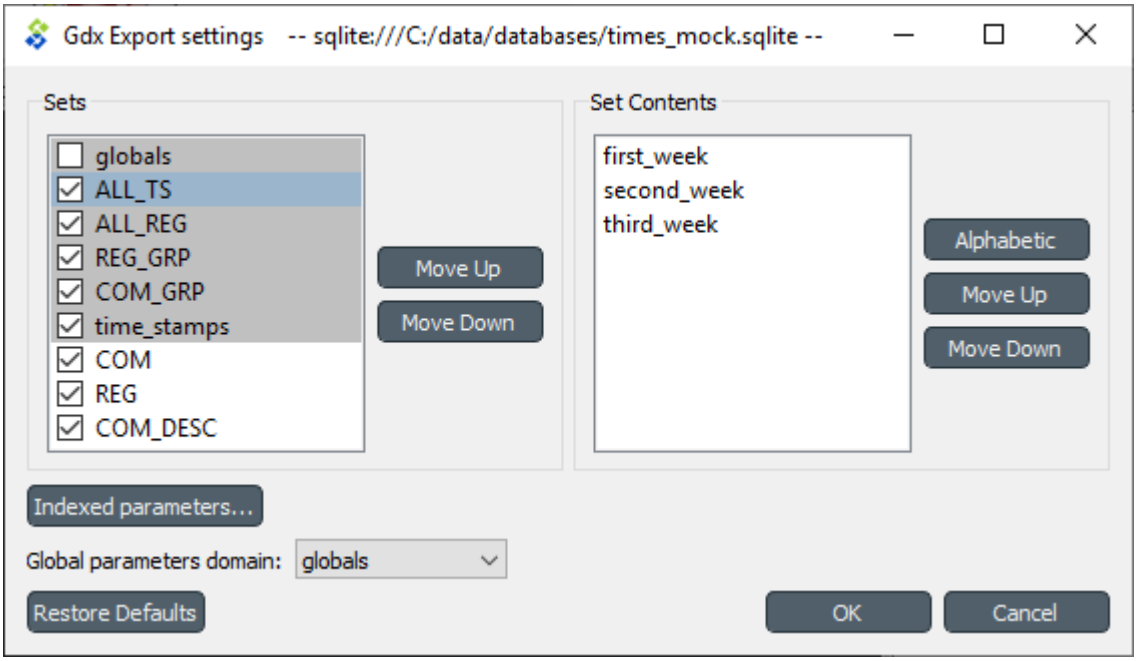
Some GAMS models need their data to be in a specific order in the `.gdx`. This is not directly supported by the database. Rather, user has to specify the desired exporting order using the *GdxExporter* item's settings.

14.3.1 GdxExporter Project Item

The image below shows the properties dock of *GdxExporter* with two *Data Sources* connected to it.



For each connected *Data Store* a box with the database's URL and export file name field is shown on the dock. The *Settings...* buttons open *Gdx Export settings* windows to allow editing database specific export parameters such as the order in which entities are exported from the database.



The *Gdx Export settings* window (see above) contains a *Sets* list which shows all GAMS sets (gray background) and subsets that are available in the database. The sets are exported in the order they are shown in the list. The *Move Up* and *Move Down* buttons can be used to move the selected set around. Note that you cannot mix sets with subsets so all sets always get exported before the subsets.

The checkbox next to the set name is used to control which sets are actually exported. Note that it is not possible to change this setting for certain sets. Global parameters domain is never exported, only its parameters which become GAMS scalars. Further, sets created for *Indexed paramaters* are always exported.

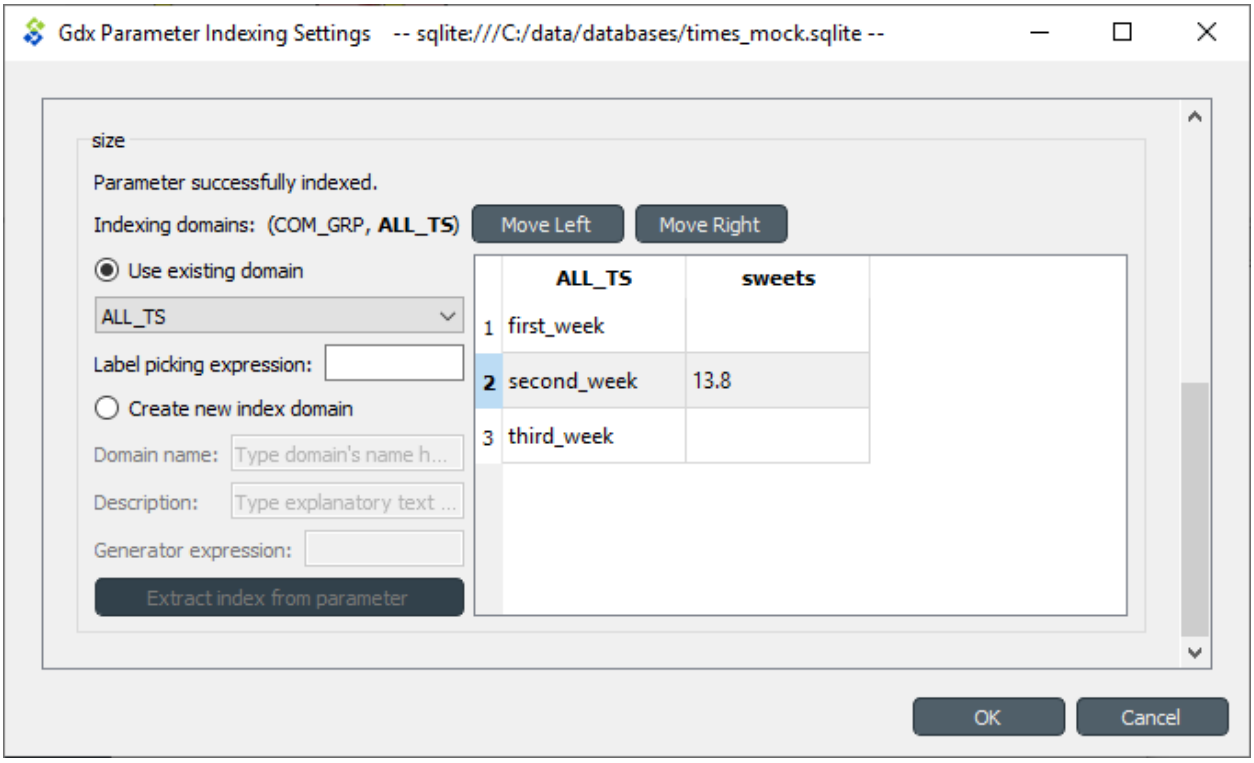
The *Set Contents* box lists the members of the selected set or subset. Their order of export can be changed the same way as with sets by *Move Up* and *Move Down*. The *Alphabetic* button sorts the members alphabetically.

Time series and time patterns cannot be exported as-is. They need to be tied up to a GAMS set. This can be achieved from the window that opens from the *Indexed parameters...* button. See the *Exporting time series and patterns* section below for more information.

Finally, one of the sets can be designated as the global parameter set. This is achieved by choosing the set's name in the *Global parameters domain* box. Note that this set is not exported, only its parameters are. They end up as GAMS scalars.

14.3.2 Exporting time series and patterns

Since GAMS has no notion of time series or time patterns these types need special handling when exported to a `.gdx` file. Namely, the time stamps or time periods (i.e. parameter indexes) need be available as GAMS sets in the exported file. It is possible to use an existing set or create a new one for this purpose. The functionality is available in *Gdx Parameter Indexing Settings* window accessible from the *Indexed Parameters...* button.

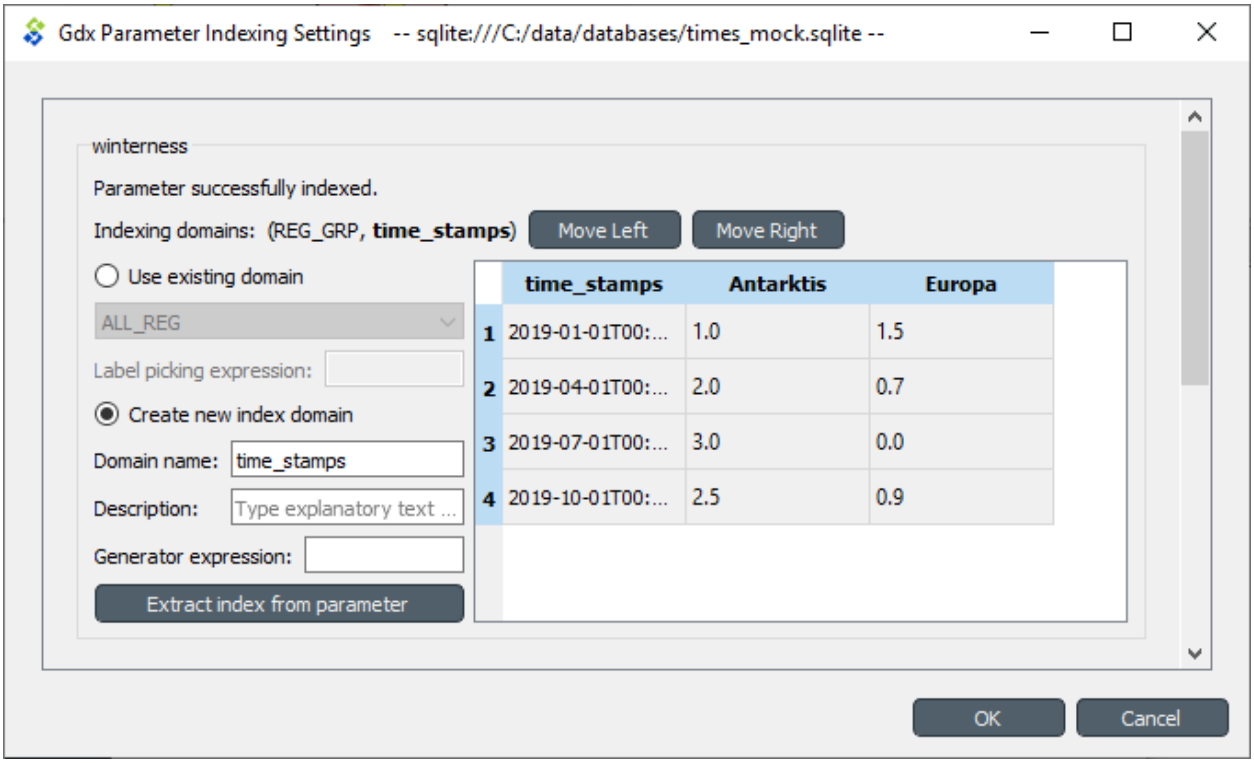


The above figure shows the indexing settings when an existing GAMS set is used to replace the original time stamps of a time series in a parameter called 'size'. The choice between using an existing set or creating a new one can be changed by the *Use existing domain* and *Create new index domain* radio buttons. When using an existing set it is selected by the combobox. In the above figure, *ALL TS* set is used for indexing.

In case of existing set it is possible that not all the set's contents are used for indexing. The table occupying the right side of the above figure shows which of the set's keys index which parameter values. The first column contains the keys of the currently selected set whereas the other columns contain the parameter's values, one column for each object that has the parameter. Selecting and deselecting rows in the table changes the indexing as only the keys on selected rows are used to index the parameter. **Shift**, **ctrl** and **ctrl-A** help in manual selection. If the selected indexes have certain pattern it might be useful to utilize the *Label picking expression* field which selects the set keys using a Python expression returning a boolean value. Some examples:

Expression	Effect
`i == 3`	Select the third row only
`i % 2 == 0`	Select even rows
`(i + 1) % 2 == 0 and i != 9`	Select odd rows except row 9

The *Indexing domains* list allows to shuffle the order of the parameter's dimensions. The **bold** dimension is the new dimension that is added to the parameter. It can be moved around by the *Move Left* and *Move Right* buttons.



It is possible to create a new indexing set by choosing *Create new index domain* as shown in the figure above. *Domain name* is mandatory for the new domain. A *Description* can also be provided but it is optional. There are two options to generate the index keys: extract the time stamps or time periods from the parameter itself or generate them using a Python expression. The *Extract index from parameter* button can be used to extract the keys from the parameter. The *Generator expression* field, on the other hand, is used to generate index keys for the new set. The expression should return Python object that is convertible to string. Below are some example expressions:

Expression	Keys
`i`	1, 2, 3,...
`f"{i - 1:04}"`	0000, 0001, 0002,...
`f"T{i:03}"`	T001, T002, T003,...

CHAPTER

FIFTEEN

SPINE ENGINE SERVER

15.1 Setting up Spine Engine Server

1. Make a new environment for Spine Engine Server
   - Make a miniconda environment & activate it
   - Clone and checkout spine-engine
   - cd to spine-engine repo root, run:

     ```
     pip install -e .
     ```

   - Clone and checkout spine-items
   - cd to spine-items repo root, run:

     ```
     pip install --no-deps -e .
     ```

2. Create security credentials (optional)
   - cd to <repo_root>/spine_engine/server/
   - Create security certificates by running *python certificate_creator.py*
   - The certificates are created into <repo_root>/spine_engine/server/certs/ directory.
   - Configure allowed endpoints by creating file *<repo_root>/spine_engine/server/connectivity/certs/allowEndpoints.txt*
   - Add IP addresses of the remote end points to the file
3. Install IPython kernel spec (*python3*) to enable Jupyter Console execution of Python Tools
   - Run:

     ```
     python -m pip install ipykernel
     ```

4. Install Julia 1.8
   - Download from https://julialang.org/downloads/ or run *apt-get install julia* on Ubuntu
5. Install IJulia kernel spec (*julia-1.8*) to enable Jupyter Console execution of Julia tools
   - Open Julia REPL and press *]* to enter pkg mode. Run:

     ```
     add IJulia
     ```

   - This installs *julia-1.8* kernel spec to *~/.local/share/jupyter/kernels* on Ubuntu or to *%APPDATA%jupyterkernels* on Windows

6. Start Spine Engine Server
   - cd to <repo_root>/spine_engine/server/
   - Without security, run:

```
python start_server.py 50001
```

   - where 50001 is the server port number.
   - With Stonehouse security, run:

```
python start_server.py 50001 StoneHouse <repo_root>/spine-engine/server/
↪connectivity/certs
```

   - where, 50001 is the server port number, StoneHouse is the security model, and the path is the folder containing the security credentials.

Note: Valid port range is 49152-65535.

15.2 Setting up Spine Toolbox (client)

1. (Optional) If server is started using StoneHouse security, copy security credentials from the server to some directory. Server's secret key does not need to be copied.
2. Start Spine Toolbox and open a project
3. Open the **Engine** page in application settings (**File->Settings**)
   - Enable remote execution from the checkbox (Enabled)
   - Set up the Spine Engine Server settings (host, port, security model, and security folder). Host is 127.0.0.1 when the Server runs on the same computer as the client.
   - Click Ok, to close and save the new Settings
4. Click Play to execute the project.

CHAPTER

SIXTEEN

TERMINOLOGY

Here is a list of definitions related to Spine project, SpineOpt.jl, and Spine Toolbox.

- **Arc** Graph theory term. See *Connection*.
- **Case study** Spine project has 13 case studies that help to improve, validate and deploy different aspects of the SpineOpt.jl and Spine Toolbox.
- **Connection** an arrow on Spine Toolbox Design View that is used to connect project items to each other to form a DAG.
- **Data Connection** is a project item used to store a collection of data files that may or may not be in Spine data format. It facilitates data transfer from original data sources e.g. spreadsheet files to Spine Toolbox. The original data source file does not need to conform to the format that Spine Toolbox is capable of reading, since there we can use an interpreting layer (Importer) between the raw data and the Spine format database (Data Store).
- **Data Package** is a data container format consisting of a metadata descriptor file (`datapackage.json`) and resources such as data files.
- **Data sources** are all the original, unaltered, sources of data that are used to generate necessary input data for Spine Toolbox tools.
- **Data Store** is a project item. It's a Spine Toolbox internal data container which follows the Spine data model. A data store is implemented using a database, it may be, for example, an SQL database.
- **Design View** A *sub-window* on Spine Toolbox main window, where project items and connections are visualized.
- **Direct predecessor** Immediate predecessor. E.g. in DAG *x->y->z*, direct predecessor of node *z* is node *y*. See also predecessor.
- **Direct successor** Immediate successor. E.g. in DAG *x->y->z*, direct successor of node *x* is node *y*. See also successor.
- **Directed Acyclic Graph (DAG)** Finite directed graph with no directed cycles. It consists of vertices and edges. In Spine Toolbox, we use project items as vertices and connections as edges to build a DAG that represents a data processing chain (workflow).
- **Edge** Graph theory term. See *Connection*
- **GdxExporter** is a project item that allows exporting a Spine data structure from a Data Store into a .gdx file which can be used as an input file in a Tool.
- **Importer** is a project item that can be used to import data from e.g. an Excel file, transform it to Spine data structure, and into a Data Store.
- **Node** Graph theory term. See *Project item*.
- **Predecessor** Graph theory term that is also used in Spine Toolbox. Preceding project items of a certain project item in a DAG. For example, in DAG *x->y->z*, nodes *x* and *y* are the predecessors of node *z*.

- **Project** in Spine Toolbox consists of project items and connections, which are used to build a data processing chain for solving a particular problem. Data processing chains are built and executed using the rules of Directed Acyclic Graphs. There can be any number of project items in a project.
- **Project item** Spine Toolbox projects consist of project items. Project items together with connections are used to build Directed Acyclic Graphs (DAG). Project items act as nodes and connections act as edges in the DAG. See *Project Items* for an up-to-date list on project items available in Spine Toolbox.
- **Scenario** A scenario is a meaningful data set for the target tool.
- **Spine data structure** Spine data structure defines the format for storing and moving data within Spine Toolbox. A generic data structure allows representation of many different modelling entities. Data structures have a class defining the type of entity they represent, can have properties and can be related to other data structures. Spine data structures can be manipulated and visualized within Spine Toolbox while SpineOpt.jl will be able to directly utilize as well as output them.
- **SpineOpt.jl** An interpreter, which formulates a solver-ready mixed-integer optimization problem based on the input data and the equations defined in the SpineOpt.jl. Outputs the solver results.
- **Source directory** In context of Tool specifications, a source directory is the directory where the main program file of the Tool specification is located. This is also the recommended place for saving the Tool specification file (.json).
- **Successor** Graph theory term that is also used in Spine Toolbox. Following project items of a certain project item in a DAG. For example, in DAG *x->y->z*, nodes *y* and *z* are the successors of node *x*.
- **Tool** is a project item that is used to execute Python, Julia, GAMS, executable scripts, or simulation models. This is done by creating a Tool specification defining the script or program the user wants to execute in Spine Toolbox. Then you need to attach the Tool specification to a Tool project item. Tools can be used to execute a computational process or a simulation model, or it can also be a process that converts data or calculates a new variable. In general, Tools may take some data as input and produce an output.
- **Tool specification** is a JSON structure that contains metadata required by Spine Toolbox to execute a computational process or a simulation model. The metadata contains; type of the program (Python, Julia, GAMS, executable), main program file (which can be e.g. a Windows batch (.bat) file or for Python scripts this would be the .py file where the __main__() method is located), All additional required program files, any optional input files (e.g. data), and output files. Also any command line arguments can be defined in a Tool specification. SpineOpt.jl is a Tool specification from Spine Toolbox's point-of-view.
- **Use case** Potential way to use Spine Toolbox. Use cases together are used to test the functionality and stability of Spine Toolbox and SpineOpt.jl under different potential circumstances.
- **Vertice** Graph theory term. See *Project item*.
- **View** A project item that can be used for visualizing project data.
- **Work directory** Tool specifications can be executed in *Source directory* or in *work directory*. When a Tool specification is executed in a work directory, Spine Toolbox creates a new *work* directory, copies all required and optional files needed for running the Tool specification to this directory and executes it there. After execution has finished, output or result files can be copied into a timestamped (archive) directory from the work directory.

CHAPTER

SEVENTEEN

DEPENDENCIES

Spine Toolbox requires Python 3.7 or Python 3.8. Python 3.9 is not supported yet.

The dependencies have been split to required packages and development packages. The required packages must be installed for the application to start. The development packages contain tools that are recommended for developers. If you want to deploy the application yourself by using the provided *cx_Freeze_setup.py* file, you need to install the *cx_Freeze* package (v6.6 or newer recommended).

At the moment, Spine Toolbox depends on four main packages (*spinetoolbox*, *spine-engine*, *spine-items*, and *spinedb-api*) developed in Spine project. For version number limitations, please see *requirements.txt* and *setup.py* files in *spinetoolbox*, *spine-engine*, *spine-items*, and *spinedb-api* packages

17.1 Dependencies by package

17.1.1 spinetoolbox

Package name	License
spinedb-api	LGPL
spine_engine	LGPL
spine_items*	LGPL
pyside2	LGPL
datapackage	MIT
jupyter-client	BSD
qtconsole	BSD
sqlalchemy	MIT
numpy	BSD
matplotlib	BSD
scipy	BSD
networkx	BSD
cx_Oracle	BSD
pandas	BSD
pymysql	MIT
pyodbc	MIT
psycopg2	LGPL
jill	MIT

* spine-items is not a 'hard' requirement of Spine Toolbox. The app does start without spine-items but the features in that case are quite limited.

17.1.2 spinedb-api

Package name	License
sqlalchemy	MIT
alembic	MIT
faker	MIT
python-dateutil	PSF
numpy	BSD
openpyxl	MIT/Expat
gdx2py	MIT
ijson	BSD

17.1.3 spine-engine

Package name	License
spinedb-api	LGPL
dagster	Apache-2.0
sqlalchemy	MIT
numpy	BSD
datapackage	MIT

17.1.4 spine-items

Package name	License
spinetoolbox	LGPL
spinedb-api	LGPL
spine-engine	LGPL

17.2 Development packages

Below is a list of development packages in *dev-requirements.txt*. Sphinx and sphinx_rtd_theme packages are needed for building the user guide. Black is used for code formatting while pylint does linting. Pre-commit hook enables automatic code formatting at git commit.

Package name	License
black	MIT
pre-commit	MIT
pyYAML	GPL
pylint	GPL
sphinx	BSD
sphinx_rtd_theme	MIT
recommonmark	MIT
sphinx-autoapi	MIT

CHAPTER

EIGHTEEN

CONTRIBUTION GUIDE FOR SPINE TOOLBOX

All are welcome to contribute! This guide is based on a set of best practices for open source projects [JF18].

18.1 Reporting Bugs

18.1.1 Due Diligence

Before submitting a bug report, please do the following:

Perform basic troubleshooting steps.

1. **Make sure you're on the latest version.** If you're not on the most recent version, your problem may have been solved already! Upgrading is always the best first step.
2. **Try older versions.** If you're already on the latest release, try rolling back a few minor versions (e.g. if on 1.7, try 1.5 or 1.6) and see if the problem goes away. This will help the devs narrow down when the problem first arose in the commit log.
3. **Try switching up dependency versions.** If you think the problem may be due to a problem with a dependency (other libraries, etc.). Try upgrading/downgrading those as well.
4. **Search the project's bug/issue tracker to make sure it's not a known issue.** If you don't find a pre-existing issue, consider checking with the maintainers in case the problem is non-bug-related. Spine Toolbox issue tracker is here.

18.1.2 What to Put in Your Bug Report

Make sure your report gets the attention it deserves: bug reports with missing information may be ignored or punted back to you, delaying a fix. The below constitutes a bare minimum; more info is almost always better:

1. What version of the Python interpreter are you using? E.g. Python 2.7.3, Python 3.6?
2. What operating system are you on? Windows? (Vista, 7, 8, 8.1, 10). 32-bit or 64-bit? Mac OS X? (e.g. 10.7.4, 10.9.0) Linux (Which distro? Which version of that distro? 32 or 64 bits?) Again, more detail is better.
3. Which version or versions of the software are you using? If you have forked the project from Git, which branch and which commit? Otherwise, supply the application version number (Help->About menu). Also, ideally you followed the advice above and have ruled out (or verified that the problem exists in) a few different versions.
4. How can the developers recreate the bug? What were the steps used to invoke it. A screenshot demonstrating the bug is usually the most helpful thing you can report (if applicable) Relevant output from the Event Log or debug messages from the console of your run, should also be included.

18.2 Feature Requests

The developers of Spine Toolbox are happy to hear new ideas for features or improvements to existing functionality. The format for requesting new features is free. Just fill out the required fields on the issue tracker and give a description of the new feature. A picture accompanying the description is a good way to get your idea into development faster. But before you make a new issue, check that there isn't a related idea already open in the issue tracker. If you have an idea on how to improve an existing idea, just join the conversation.

18.3 Submitting features/bugfixes

If you feel like you can fix a bug that's been bothering you or you want to add a new feature to the application but the devs seem to be too busy with something else, please follow the instructions in the following sections on how to contribute code.

18.3.1 Coding Style

Follow the style you see used in the repository! Consistency with the rest of the project always trumps other considerations. It doesn't matter if you have your own style or if the rest of the code breaks with the greater community - just follow along.

Spine Toolbox coding style follows PEP-8 style guide for Python code with the following variations:

- Maximum line length is 120 characters. Longer lines are acceptable for a good reason.
- Google style docstrings with the title and input parameters are required for all classes, functions, and methods. For small functions or methods only the summary is necessary. Return types are highly recommended but not required if it is obvious what the function or method returns.
- Use double-quoted strings instead of single-quoted strings (e.g. `"hello"`).
- Other deviations from PEP-8 can be discussed.

18.3.2 Commit messages

The commit message should tell *what* was changed and *why*. Details on *how* it was done can usually be left out, if the code itself is self-explanatory (remember source comments too!). Separate the subject line from the body with a blank line. The subjet line (max. 50 chars) should explain in condensed form what happened using imperative mood, i.e. using verbs like 'change', 'fix' or 'add'. Start the subject line with a capital letter. Do not use the issue number on the subject line, as it does not tell much to a person who's not aware of that particular issue. For more info see Chris Beams' 'Seven rules of of a great Git commit message' [CB14].

A good example (insipred by [CB14])

```
Fix bugs when updating parameters in foo and bar

Body of the commit message starts after a blank line. Explain here in more
detail the reasons why you made the change, how things worked before and how they work␣
↪now.
Also explain why

You can use hyphens to make bulleted lists:
- Foo was added because of bar
```

(continues on next page)

(continued from previous page)

```
- Baz was not used so it was deleted

Add references to issue tracker (if any) at the end.

Solves: #123
See also: #456, #789
```

18.3.3 Contributing to the User Guide

Spine Toolbox uses Sphinx to create HTML pages from restructured text (.rst) files. The .rst files are plain text files that are formatted in a way that Sphinx understands and is able to turn them into HTML. Please see this brief introduction for more on reStructured text. You can modify the existing or create new .rst files into `docs/source` directory. When you are done editing, run `bin/build_doc.bat` on Windows or `bin/build_doc.py` on other systems to build the HTML pages to check the result before making a commit. The created pages are found in `docs/build/html` directory. After a commit, the User Guide is built automatically by readthedocs.org. The latest User Guide is available in https://spine-toolbox.readthedocs.io/en/latest/.

18.3.4 Contributing to the Spine Toolbox Graphical User Interface

If you want to change or add new widgets into the application, you need to use the `bin\build_ui.bat` (Windows) or `bin/build_ui.py` (other systems) scripts. The main design of the widgets should be done with Qt Designer (`designer.exe` or `designer`) that is included with PySide2. The files produced by Qt Designer are XML files (.ui). You can also embed graphics (e.g. icons, logos, etc.) into the application by using Qt Designer. When you are done modifying widgets in the designer, you need to run the `build_ui` script for the changes to take effect. This script uses tools provided in the PySide2 package to turn .ui files into Python files, in essence rebuilding the whole Spine Toolbox user interface.

Styling the widgets should be done with Qt Style Sheets in code. Please avoid using style sheets in Qt Designer.

18.3.5 Version Control Branching

Always make a new branch for your work, no matter how small. This makes it easy for others to take just that one set of changes from your repository, in case you have multiple unrelated changes floating around. A corollary: don't submit unrelated changes in the same branch/pull request! The maintainer shouldn't have to reject your awesome bugfix because the feature you put in with it needs more review.

Name your new branch descriptively, e.g. `issue#XXX-fixing-a-serious-bug` or `issue#ZZZ-cool-new-feature`. New branches should in general be based on the latest `master` branch. In case you want to include a new feature still in development, you can also start working from its branch. The developers will backport any relevant bug-fixes to previous or upcoming releases under preparation.

If you need to use code from an upstream branch, please use git-rebase *if you have not shared your work with others yet*. For example: You started working on an issue, but now the upstream branch (`master`) has some new commits you would like to have in your branch too. If you have not yet pushed your branch, you can now rebase your changes on top of the upstream branch:

```
$ git pull origin master:master
$ git checkout my_branch
$ git rebase master
```

Avoid merging the upstream branch to your issue branch if it's not necessary. This will lead to a more linear and cleaner history.

Finally, make a pull request from your branch so that the developers can review your changes. You might be asked to make additional changes or clarifications or add tests to prove the new feature works as intended.

18.3.6 Test-driven development is your friend

Any bug fix that doesn't include a test proving the existence of the bug being fixed, may be suspect. Ditto for new features that can't prove they actually work.

It is recommended to use test-first development as it really helps make features better designed and identifies potential edge cases earlier instead of later. Writing tests before the implementation is strongly encouraged.

See *Unit testing guidelines* for more information.

18.3.7 Full example

Here's an example workflow. Your username is `yourname` and you're submitting a basic bugfix.

Preparing your Fork

1. Click 'Fork' on Github, creating e.g. `yourname/Spine-Toolbox`
2. Clone your project: `git clone git@github.com:yourname/Spine-Toolbox`
3. `cd Spine-Toolbox`
4. Create a virtual environment and install requirements
5. Create a branch: `git checkout -b foo-the-bars master`

Making your Changes

1. Add an entry to CHANGELOG.md.
2. Write tests expecting the correct/fixed functionality; make sure they fail.
3. Hack, hack, hack.
4. Run tests again, making sure they pass.
5. Commit your changes: `git commit -m "Foo the bars"`

Creating Pull Requests

1. Push your commit to get it back up to your fork: `git push origin HEAD`
2. Visit Github, click handy 'Pull request' button that it will make upon noticing your new branch.
3. In the description field, write down issue number (if submitting code fixing an existing issue) or describe the issue + your fix (if submitting a wholly new bugfix).
4. Hit 'submit'! And please be patient - the maintainers will get to you when they can.

18.4 References

CHAPTER

NINETEEN

DEVELOPER DOCUMENTATION

Here you can find developer specific documentation on Spine Toolbox.

19.1 UI guidelines

19.1.1 Keyboard shortcuts

Qt has a list of 'standard' keyboard shortcuts which can be used for inspiration.

- **F2**: edit current value in-place
- **Alt + F2**: open separate editor (e.g. Parameter value editor)
- **F3**: search
- **Alt-F4**: quit, close without saving changes
- **Esc**: close, exit without saving changes
- **Ctrl + Enter**: accept dialog

19.1.2 Action names

- **Edit…** should open an external editor, e.g. Parameter value editor in Database editor.

19.2 Unit testing guidelines

19.2.1 Test modules, directories

Spine project uses Python standard unittest framework for testing. The tests are organized into Python modules starting with the prefix `test_` under `<project root>/tests/`. The structure of `tests/` mirrors that of the package being tested. Note that all subdirectories containing test modules under `tests/` must have an (empty) `__init__.py` which makes them part of the project's test package.

While there are no strict rules on how to name the individual test modules except for the `test_` prefix, `test_<module_name>.py` is preferred.

19.2.2 Running the tests

Tests are run as a GitHub action whenever a branch is pushed to GitHub. This process is configured by `<project root>/.github/workflows/unittest_runner.yml`

To execute the tests manually, run `python -munittest discover` in project's root.

19.2.3 Helpers

`mock_helpers` module in Toolbox's test package contains some helpful functions. Especially the methods to create mock `ToolboxUI` and `SpineToolboxProject` objects come very handy.

When instantiation of `QWidget` (this includes all GUI testing) is needed, Qt's main loop must be running during testing. This can be achieved by e.g. the `setUpClass` method below:

```
@classmethod
def setUpClass(cls):
    if not QApplication.instance():
        QApplication()
```

Sometimes an in-memory database can be handy because it does not require a temporary files or directories and it may be faster than an `.sqlite` file. To create an in-memory database, use `sqlite://` as the URL:

```
db_map = DiffDatabaseMapping("sqlite://", create=True)
```

Unfortunately, it is not possible to refer to the created database with the same URL prohibiting multiple database maps the access to the same in-memory database.

19.3 Execution tests

Toolbox contains *execution tests* that test entire workflows in the headless mode. The tests can be found in `<toolbox repository root>/execution_tests/`. Execution tests are otherwise normal Toolbox projects except that the project root directories contain `__init__.py` and `execution_test.py` files. `__init__.py` makes the directory part of the execution test suite while `execution_test.py` contains actual test code. The tests utilize Python's `unittest` package so the test code is practically identical to any unit tests in Toolbox.

19.3.1 Executing the tests

Tests are run as a GitHub action whenever a branch is pushed to GitHub. This process is configured by `<project root>/.github/workflows/executiontest_runner.yml`

To execute the tests manually, run `python -munittest discover --pattern execution_test.py` in project's root.

19.4 Project item development

This document discusses the basics of *project item* development: what is required make one, how items interact with the Toolbox GUI and how they are executed.

The core of every project item consists of two classes: a *static* project item class which is responsible for integrating the item with the Toolbox GUI and an *executable* class which does the item's 'thing' and exists only during execution in Spine Engine. Some additional classes are needed for Toolbox to be able to instantiate project items and to communicate with the user via the Toolbox GUI.

Specifications are a way to make the settings of an item portable across projects. In a sense a specification is a template that can specialize an item for a specific purpose such as a Tool that runs certain model with known inputs an outputs. Items that support specifications need to implement some additional methods and classes.

19.4.1 Getting started

Probably the most convenient way to start developing a new project item is to work with a copy of some simple project item. For example, **View** provides a good starting point.

Project items are mostly self-contained Python packages. It is customary to structure the project item packages like the Toolbox itself: `mvcmodels` submodule for Qt's models, `ui` module for automatically generated UI forms and `widgets` for widgets' business logic. However, the only actual requirement is that Toolbox expects to find the item's factory and item info classes in the package's root modules as well as an `executable_item` module.

19.4.2 Item info

A subclass of `spine_engine.project_item.project_item_info.ProjectItemInfo` must be found in one of the root modules of an item's package. It is used by Toolbox to query the *type* and *category* of an item. Type identifies the project item while category is used by the Toolbox GUI to group project items with similar function. Categories are currently fixed and can be checked from `spine_items.category`.

19.4.3 Item Factory

The details of constructing a project item and related objects have been abstracted away from Toolbox by a factory that must be provided by every project item in a root module of the item's package. The factory is a subclass of *`spinetoolbox.project_item.project_item_factory.ProjectItemFactory`*. Note that methods in the factory that deal with specifications need to be implemented only by items that support them.

19.4.4 Executable item

A project item must have a root module called `executable_item` that contains a class named `ExecutableItem` which is a subclass of `spine_engine.project_item.executable_item_base.ExecutableItemBase` `ExecutableItem` acts as an access point to Spine Engine and contains the item's execution logic.

19.4.5 Toolbox side project item

A project item must subclass *spinetoolbox.project_item.project_item.ProjectItem* and return the subclass in its factory's `item_class()` method. Also `make_item()` must return an instance of this class. This class forms the core of integrating the item with Toolbox.

19.4.6 Specifications

Items that support specifications need to subclass `spine_engine.project_item.project_item_specification_factory.ProjectItemSpecificationFactory` which provides an access point to Toolbox and Spine Engine to generate specifications. The factory must be called `SpecificationFactory` and be placed in `specification_factory` module under item package's root. The specification itself should be a subclass of `spine_engine.project_item.project_item_specification.ProjectItemSpecification`.

19.4.7 Toolbox GUI integration

`ProjectItemFactory.icon()` returns a URL to the item's icon resource. This is the item's 'symbol' shown e.g. on the main toolbar of Toolbox. It should not be confused with the actual icon on Design view which in turn is a subclass of `spinetoolbox.project_item.project_item_icon.ProjectItemIcon` and is returned by `ProjectItemFactory.make_icon()`.

When creating a new item on the Design view Toolbox shows the *Add item dialog* it gets from `ProjectItemFactory.make_add_item_widget()`. Toolbox provides *spinetoolbox.widgets.add_project_item_widget.AddProjectItemWidget* which is a general purpose widget for this purpose though project items are free to implement their own widgets as needed.

Once the item is on the Design view, the main interaction with it goes through the properties widget which is created by `ProjectItemFactory.make_properties_widget()`. The properties widget should have all controls needed to set up the item.

19.4.8 Saving and restoring project items

Project items are saved in JSON format as part of the `project.json` file. Item saving is handled by `ProjectItem.item_dict()` which should return a JSON compatible `dict` and contain at least the information returned by the base class method.

File system paths are handled specifically during saving: all paths outside the project directory should be absolute while the paths in the project directory should be relative. This is to enable self-contained projects which include all needed files and can be easily transferred from system to system. As such, paths are saved as special dictionaries. `spine_engine.utils.serialization.serialize_path()`, `spine_engine.utils.serialization.serialize_url()` and `spine_engine.utils.serialization.deserialize_path()` help with dealing with the paths.

`ProjectItem.from_dict()` is responsible for restoring a saved project item from the dictionary. `ProjectItem.parse_item_dict()` can help to deserialize the basic data needed by the base class.

19.4.9 Passing data between items: resources

Project items share data by files or via databases. One item writes a file which is then read by another item. **Project item resources** are used to communicate the URLs of these files and databases.

Resources are instances of the `spine.engine.project_item.project_item_resource.ProjectItemResource` class.

Both static items and their executable counterparts pass resources. The major difference is that static item's may pass resource *promises* such as files that are generated during the execution. The full path to the promised files or even their final names may not be known until the items are executed.

During execution resources are propagated only to item's *direct* predecessors and successors. Static items offer their resources to direct successors only. Resources that are communicated to successor items are basically output files that the successor items can use for input. Currently, the only resource that is propagated to predecessor items is database URLs by Data Store project items. As Data Stores leave the responsibility of writing to the database to other items it has to tell these items where to write their output data.

The table below lists the resources each project item type provides during execution.

Item	Notes	Provides to predecessor	Provides to successor
Data Connection	[1]	n/a	File URLs
Data Store	[2]	Database URL	Database URL
Data Transformer	[3]	n/a	Database URL
Exporter		n/a	File URLs
GdxExporter		n/a	File URLs
Gimlet		n/a	Resources from predecessor
Importer		n/a	n/a
Tool	[4]	n/a	File URLs
View		n/a	n/a

The table below lists the resources that might be used by each item type during execution.

Item	Notes	Accepts from predecessor	Accepts from successor
Data Connection		n/a	n/a
Data Store		n/a	n/a
Data Transformer		Database URL	n/a
Exporter		Database URL	n/a
GdxExporter		Database URL	n/a
Gimlet	[5]	File URLs, database URLs	Database URLs
Importer	[6]	File URLs	Database URL
Tool	[7]	File URLs, database URLs	Database URLs
View		Database URLs	n/a

[1] Data connection provides paths to local files.

[2] Data Store provides a database URL to direct successors and predecessors. Note, that this is the only project item that provides resources to it's predecessors.

[3] Data Transformer provides its predecessors' database URLs modified by transformation configuration embedded in the URL.

[4] Tool's output files are specified by a *Tool specification*.

[5] Gimlet's resources can be passed to the command as command line arguments but are otherwise ignored.

[6] Importer requires a database URL from its successor for writing the mapped data. This can be provided by a Data Store.

[7] *Tool specification* specifies tool's optional and required input files. Database URLs can be passed to the tool *program* via command line arguments but are otherwise ignored by the Tool project item. Currently, there is no mechanism to know if a URL is actually required by a tool *program*. For more information, see *Tool specification editor*.

19.4.10 Execution

Spine Engine instantiates the executable items in a DAG before the execution starts. Then, Engine declares forward and backward resources for each item using `ExecutableItemBase.output_resources()`. During execution, `ExecutableItemBase.execute()` is invoked with lists of available resources if an item is selected for execution. Otherwise, `ExecutableItemBase.exclude_execution()` is called.

19.5 Publishing to PyPI

This document describes the prerequisites and workflow to publish Spine Toolbox to The Python Package Index (PyPI).

First, make sure you have all the developer packages installed by calling

```
$ pip install --upgrade -r dev-requirements.txt
```

inside your Python environment.

The most convenient order in which Spine packages should be published is `spinedb_api`, then `spine_engine`, `spine_items`, and last `spinetoolbox`.

For each of the packages, make sure you are on the `master` branch and the repositories are up-to-date. Also, test that Toolbox works as expected.

Starting from `spinedb_api`, the following steps should be taken.

19.5.1 1. Update dependencies

This step is skipped for `spinedb_api` as it does not depend on other packages.

Update `setup.cfg` or `setup.py` files so that they require the latest dependencies. For example, if you just published `spinedb_api` 0.99.0, ensure that `spine_engine`'s `setup.py` includes the line

```
"spinedb_api>=0.99.0",
```

Note: `spine_items` and `spinetoolbox` depend circularly on each other. When updating `spine_items`'s `setup.cfg`, you should update the `spinetoolbox` entry as well even though that version will be published later.

Finally, please remember to commit and push your changes before continuing.

19.5.2 2. Tag the git revision

Tag the code revision with

```
$ git tag --message "Version x.y.z" <version number>
```

You can find the current version in `<package>/version.py` files except for `spinedb_api` which stores the version in `spinedb_api/__init__.py`.

Note that you should drop the release level part of the version number, e.g. 0.11.0.dev0 becomes 0.11.0 for the purpose of the git tag.

19.5.3 3. Make version final

The release level of Spine packages is 'dev' by default but we want to upload final versions to PyPI. Remove the `.dev0` part of the package's version string in `version.py` (`__init__.py` in case of `spinedb_api`) or replace `dev` by `final` if you are working with `spinetoolbox`.

Do not commit this change to git. The version number will be updated again in the last step.

19.5.4 4. Clean up previous release builds

Delete the `build/` and `dist/` directories in the repository root if they exist. The directories will be autogenerated in the build step.

Another option to clean previous builds is with

```
$ python setup.py clean --all
```

19.5.5 5.Build

Build a source distribution archive and a wheel package with

```
$ python setup.py sdist bdist_wheel
```

This will create distribution files under the `dist/` and `build/` directories.

Please remember to clean up between subsequent builds per the instructions in the previous step.

19.5.6 6. Upload

Before making a real upload, please test using TestPyPI which is a separate instance from the real index server. Once a version has been uploaded to PyPI, it cannot be reverted or modified.

Register an account and ask some of the owners of the Spine Toolbox package (or other relevant package) to add you as a maintainer.

Upload the distribution using

```
$ twine upload --repository testpypi dist/*
```

See Using TestPyPI for more information. To avoid entering your username and password every time, see Keyring support in twine documentation.

If everything went smoothly, you are ready to upload the real index. Again, you need to register to PyPI and ask to become a maintainer of the package you want to upload to. Upload the distribution using

```
$ twine upload dist/*
```

19.5.7 7. Bump version number

Now that the package has been released to PyPI, it is time to update the version to the next development version. Bump the number in `version.py` (`__init__.py` for `spinedb_api`) to the next appropriate one and append `.dev0` to the version string (replace `final` by `dev` for `spinetoolbox`).

Do not forget to push the changes.

19.5.8 8. Rinse and repeat

Switch to the next Spine package and start over from step 1 unless you just finished with `spinetoolbox`. In that case, congratulations! You have just released the Spine project to PyPI.

CHAPTER

TWENTY

API REFERENCE

This page contains auto-generated API reference documentation[1].

20.1 spinetoolbox

spinetoolbox package.

20.1.1 Subpackages

spinetoolbox.mvcmodels

Modules in this package contain classes that represent Spine Toolbox's models (internal data structures) in the Model-View-Controller design pattern. The model classes define an interface that is used by views and delegates to access data in the application.

author

P. Savolainen (VTT)

date

24.9.2019

Submodules

spinetoolbox.mvcmodels.array_model

Contains model for the Array editor widget.

author

A. Soininen (VTT)

date

14.6.2019

[1] Created with sphinx-autoapi

Module Contents

Classes

ArrayModel	Model for the Array parameter_value type.

class spinetoolbox.mvcmodels.array_model.**ArrayModel**(*parent*)

Bases: PySide2.QtCore.QAbstractTableModel

Model for the Array parameter_value type.

Even if the array is empty this model's rowCount() will still return 1. This is to show an empty row in the table view.

Parameters
parent (*QObject*) – parent object

array()

Returns the array modeled by this model.

batch_set_data(*indexes*, *values*)

Sets data at multiple indexes at once.

Parameters

- **indexes** (*list of QModelIndex*) – indexes to set
- **values** (*list of str*) – values corresponding to the indexes

columnCount(*parent=QModelIndex()*)

Returns 2.

_convert_to_data_type(*indexes*, *values*)

Converts values from string to current data type filtering failed conversions.

Parameters

- **indexes** (*list of QModelIndex*) – indexes
- **values** (*list of str*) – values to convert

Returns
indexes and converted values

Return type
tuple

data(*index*, *role=Qt.DisplayRole*)

Returns model's data for given role.

flags(*index*)

Returns table cell's flags.

headerData(*section*, *orientation*, *role=Qt.DisplayRole*)

Returns header data.

insertRows(*row*, *count*, *parent=QModelIndex()*)

Inserts rows to the array.

is_expanse_row(*row*)

Returns True if row is the expanse row.

Parameters
row (*int*) – a row

Returns
True is row is expanse row, False otherwise

Return type
bool

removeRows(*row*, *count*, *parent=QModelIndex()*)

Removes rows from the array.

reset(*value*)

Resets the model to a new array.

Parameters
value (*Array*) – a new array to model

rowCount(*parent=QModelIndex()*)

Returns the length of the array.

Note: returns 1 even if the array is empty.

set_array_type(*new_type*)

Changes the data type of array's elements.

Parameters
new_type (*Type*) – new element type

setHeaderData(*section*, *orientation*, *value*, *role=Qt.EditRole*)

setData(*index*, *value*, *role=Qt.EditRole*)

Sets the value at given index.

spinetoolbox.mvcmodels.compound_table_model

Models that vertically concatenate two or more table models.

authors

M. Marin (KTH)

date
9.10.2019

Module Contents

Classes

CompoundTableModel	A model that concatenates several sub table models vertically.
CompoundWithEmptyTableModel	A compound parameter table model where the last model is an empty row model.

class spinetoolbox.mvcmodels.compound_table_model.**CompoundTableModel**(*parent=None*, *header=None*)

Bases: *spinetoolbox.mvcmodels.minimal_table_model.MinimalTableModel*

A model that concatenates several sub table models vertically.

Initializes model.

Parameters

- **parent** (*QObject, optional*) – the parent object
- **header** (*list of str, optional*) – header labels

refreshed

map_to_sub(*index*)

Returns an equivalent submodel index.

Parameters
index (*QModelIndex*) – the compound model index.

Returns
the equivalent index in one of the submodels

Return type
QModelIndex

map_from_sub(*sub_model*, *sub_index*)

Returns an equivalent compound model index.

Parameters

- **sub_model** (*MinimalTableModel*) – the submodel
- **sub_index** (*QModelIndex*) – the submodel index.

Returns
the equivalent index in the compound model

Return type
QModelIndex

item_at_row(*row*)

Returns the item at given row.

Parameters
row (*int*) –

Returns
object

sub_model_at_row(*row*)

Returns the submodel corresponding to the given row in the compound model.

Parameters
row (*int*) –

Returns
MinimalTableModel

sub_model_row(*row*)

Calculates sub model row.

Parameters
row (*int*) – row in compound model

Returns
row in sub model

Return type
int

refresh()

Refreshes the layout by computing a new row map.

_do_refresh()

Recomputes the row and inverse row maps.

_append_row_map(*row_map*)

Appends given row map to the tail of the model.

Parameters
row_map (*list*) – tuples (model, row number)

_row_map_iterator_for_model(*model*)

Yields row map for given model. The base class implementation just yields all model rows.

Parameters
model (MinimalTableModel) –

Yields
tuple – (model, row number)

_row_map_for_model(*model*)

Returns row map for given model. The base class implementation just returns all model rows.

Parameters
model (MinimalTableModel) –

Returns
tuples (model, row number)

Return type
list

canFetchMore(*parent*)

Returns True if any of the submodels that haven't been fetched yet can fetch more.

fetchMore(*parent*)

Fetches the next sub model and increments the fetched counter.

flags(*index*)

Return index flags.

data(*index*, *role=Qt.DisplayRole*)

Returns the data stored under the given role for the item referred to by the index.

Parameters

- **index** (*QModelIndex*) – Index of item
- **role** (*int*) – Data role

Returns
Item data for given role.

rowCount(*parent=QModelIndex()*)

Returns the sum of rows in all models.

batch_set_data(*indexes*, *data*)

Sets data for indexes in batch. Distributes indexes and values among the different submodels and calls batch_set_data on each of them.

insertRows(*row*, *count*, *parent=QModelIndex()*)

Inserts count rows after the given row under the given parent. Localizes the appropriate submodel and calls insertRows on it.

removeRows(*row*, *count*, *parent=QModelIndex()*)

Removes count rows starting with the given row under parent. Localizes the appropriate submodels and calls removeRows on it.

class spinetoolbox.mvcmodels.compound_table_model.**CompoundWithEmptyTableModel**(*parent=None*, *header=None*)

Bases: *CompoundTableModel*

A compound parameter table model where the last model is an empty row model.

Initializes model.

Parameters

- **parent** (*QObject, optional*) – the parent object
- **header** (*list of str, optional*) – header labels

property single_models

property empty_model

abstract _create_empty_model()

Creates and returns an empty model.

Returns
model

Return type
EmptyRowModel

init_model()

Initializes the compound model.

Basically populates the sub_models list attribute with the result of _create_empty_model.

_connect_single_model(*model*)

Connects signals so changes in the submodels are acknowledged by the compound.

_recompute_empty_row_map()

Recomputes the part of the row map corresponding to the empty model.

_handle_empty_rows_removed(*parent*, *empty_first*, *empty_last*)

Updates row_map when rows are removed from the empty model.

_handle_empty_rows_inserted(*parent*, *empty_first*, *empty_last*)

Runs when rows are inserted to the empty model. Updates row_map, then emits rowsInserted so the new rows become visible.

_handle_single_model_about_to_be_reset(*model*)

Runs when given model is about to reset.

_handle_single_model_reset(*model*)

Runs when given model is reset.

_refresh_single_model(*model*)

_get_insert_position(*model*)

_insert_single_model(*model*)

_get_row_for_insertion(*pos*)

_insert_row_map(*pos*, *single_row_map*)

clear_model()

Clears the model.

spinetoolbox.mvcmodels.empty_row_model

Contains a table model with an empty last row.

authors

M. Marin (KTH)

date

20.5.2018

Module Contents

Classes

EmptyRowModel	A table model with a last empty row.

class spinetoolbox.mvcmodels.empty_row_model.**EmptyRowModel**(*parent=None*, *header=None*)

Bases: *spinetoolbox.mvcmodels.minimal_table_model.MinimalTableModel*

A table model with a last empty row.

Init class.

canFetchMore(*_parent*)

Return True if the model hasn't been fetched.

fetchMore(*parent*)

Fetch data and use it to reset the model.

flags(*index*)

Return default flags except if forcing defaults.

set_default_row(***kwargs*)

Set default row data.

clear()

Clear all data in model.

reset_model(*main_data=None*)

Reset model.

_handle_data_changed(*top_left*, *bottom_right*, *roles=None*)

Insert a new last empty row in case the previous one has been filled with any data other than the defaults.

removeRows(*row*, *count*, *parent=QModelIndex()*)

Don't remove the last empty row.

_handle_rows_inserted(*parent*, *first*, *last*)

Handle rowsInserted signal.

set_rows_to_default(*first*, *last=None*)

Set default data in newly inserted rows.

spinetoolbox.mvcmodels.file_list_models

Common models. Contains a generic File list model and an Item for that model. Used by the Importer, Gimlet and Tool project items but this may be handy for other project items as well.

authors

P. Savolainen (VTT), P. Vennström (VTT), A. Soininen (VTT)

date

5.6.2020

Module Contents

Classes

FileListModel	A model for files to be shown in a file tree view.
CommandLineArgItem	
NewCommandLineArgItem	
CommandLineArgsModel	
JumpCommandLineArgsModel	

class spinetoolbox.mvcmodels.file_list_models.**FileListModel**(*header_label=''*, *draggable=False*)

Bases: PySide2.QtCore.QAbstractItemModel

A model for files to be shown in a file tree view.

Parameters

- **header_label** (*str*) – header label

- **draggable** (*bool*) – if True, the top level items are drag and droppable

FileItem

PackItem

rowCount(*parent=QModelIndex()*)

columnCount(*parent=QModelIndex()*)

headerData(*section*, *orientation*, *role=Qt.DisplayRole*)

Returns header information.

data(*index*, *role=Qt.DisplayRole*)

Returns data associated with given role at given index.

flags(*index*)

mimeData(*indexes*)

resource(*index*)

Returns the resource at given index.

Parameters
index (*QModelIndex*) – index

Returns
resource

Return type
ProjectItemResource

parent(*index*)

index(*row*, *column*, *parent=QModelIndex()*)

update(*resources*)

Updates the model according to given list of resources.

Parameters
resources (*Iterable of ProjectItemResource*) – resources

duplicate_paths()

Checks if resources in the model have duplicate file paths.

Returns
set of duplicate file paths

Return type
set of str

_pack_index(*pack_label*)

Finds a pack's index in pack resources list.

Parameters
pack_label (*str*) – pack label

Returns
index to pack resources list

Return type
int

class spinetoolbox.mvcmodels.file_list_models.**CommandLineArgItem**(*text=''*, *rank=None*, *selectable=False*, *editable=False*, *drag_enabled=False*, *drop_enabled=False*)

Bases: PySide2.QtGui.QStandardItem

set_rank(*rank*)

static _make_icon(*rank=None*)

setData(*value*, *role=Qt.UserRole + 1*)

class spinetoolbox.mvcmodels.file_list_models.**NewCommandLineArgItem**

Bases: *CommandLineArgItem*

setData(*value*, *role=Qt.UserRole + 1*)

class spinetoolbox.mvcmodels.file_list_models.**CommandLineArgsModel**(*parent=None*)

Bases: PySide2.QtGui.QStandardItemModel

property args

args_updated

append_arg(*arg*)

replace_arg(*row*, *arg*)

mimeData(*indexes*)

dropMimeData(*data*, *drop_action*, *row*, *column*, *parent*)

static _reset_root(*root*, *args*, *child_params*, *has_empty_row=True*)

class spinetoolbox.mvcmodels.file_list_models.**JumpCommandLineArgsModel**(*parent=None*)

Bases: *CommandLineArgsModel*

reset_model(*args*)

canDropMimeData(*data*, *drop_action*, *row*, *column*, *parent*)

spinetoolbox.mvcmodels.filter_checkbox_list_model

Provides FilterCheckboxListModel for FilterWidget.

author

P. Vennström (VTT)

date

1.11.2018

Module Contents

Classes

SimpleFilterCheckboxListModel	Init class.
LazyFilterCheckboxListModel	Extends SimpleFilterCheckboxListModel to allow for lazy loading in synch with another model.
DataToValueFilterCheckboxListModel	Extends SimpleFilterCheckboxListModel to allow for translating internal data to a value for display role.

class spinetoolbox.mvcmodels.filter_checkbox_list_model.**SimpleFilterCheckboxListModel**(*parent*, *show_empty=True*)

Bases: PySide2.QtCore.QAbstractListModel

Init class.

Parameters
parent (*QWidget*) –

property _show_empty

property _show_add_to_selection

_SELECT_ALL_STR = (Select all)

_SELECT_ALL_FILTERED_STR = (Select all filtered)

_EMPTY_STR = (Empty)

_ADD_TO_SELECTION_STR = Add current selection to filter

reset_selection()

_handle_select_all_clicked()

_check_all_selected()

rowCount(*parent=QModelIndex()*)

data(*index*, *role=Qt.DisplayRole*)

_handle_index_clicked(*index*)

set_list(*data*, *all_selected=True*)

set_selected(*selected*, *select_empty=None*)

get_selected()

get_not_selected()

set_filter(*filter_expression*)

search_filter_expression(*item*)

set_base_filter(*condition*)

Sets the base filter. The other filter, the one that works by typing in the search bar, should be applied on top of this base filter.

Parameters

condition (*function*) – Filter acceptance condition.

apply_filter()

_remove_and_add_filtered()

_remove_and_replace_filtered()

remove_filter()

_do_add_items(*data*)

add_items(*data*, *selected=None*)

remove_items(*data*)

class spinetoolbox.mvcmodels.filter_checkbox_list_model.**LazyFilterCheckboxListModel**(*parent*, *source_model*, *show_empty=True*)

Bases: *SimpleFilterCheckboxListModel*

Extends SimpleFilterCheckboxListModel to allow for lazy loading in synch with another model.

Init class.

Parameters

- **parent** (SpineDBEditor) –
- **source_model** (CompoundParameterModel) – a model to lazily get data from

canFetchMore(*parent*)

fetchMore(*parent*)

_do_add_items(*data*)

Adds items so the list is always sorted, while assuming that both existing and new items are sorted.

class spinetoolbox.mvcmodels.filter_checkbox_list_model.**DataToValueFilterCheckboxListModel**(*parent*, *data_to_value*, *show_empty=Tr*

Bases: *SimpleFilterCheckboxListModel*

Extends SimpleFilterCheckboxListModel to allow for translating internal data to a value for display role.

Init class.

Parameters

- **parent** (SpineDBEditor) –
- **data_to_value** (*method*) – a method to translate item data to a value for display role

data(*index*, *role=Qt.DisplayRole*)

search_filter_expression(*item*)

spinetoolbox.mvcmodels.filter_execution_model

Contains FilterExecutionModel.

author

M. Marin (KTH)

date

26.11.2020

Module Contents

Classes

FilterExecutionModel

class spinetoolbox.mvcmodels.filter_execution_model.**FilterExecutionModel**

Bases: PySide2.QtCore.QAbstractListModel

_filter_consoles

reset_model(*filter_consoles*)

rowCount(*parent=QModelIndex()*)

headerData(*section*, *orientation*, *role=Qt.DisplayRole*)

data(*index*, *role=Qt.DisplayRole*)

find_index(*console_key*)

get_console(*filter_id*)

spinetoolbox.mvcmodels.indexed_value_table_model

A model for indexed parameter values, used by the parameter_value editors.

authors

A. Soininen (VTT)

date

18.6.2019

Module Contents

Classes

IndexedValueTableModel	A base class for time pattern and time series models.

Attributes

EXPANSE_COLOR

spinetoolbox.mvcmodels.indexed_value_table_model.**EXPANSE_COLOR**

class spinetoolbox.mvcmodels.indexed_value_table_model.**IndexedValueTableModel**(*value*, *parent*)

Bases: PySide2.QtCore.QAbstractTableModel

A base class for time pattern and time series models.

Parameters

- **value** (*IndexedValue*) – a parameter_value
- **parent** (*QObject*) – parent object

property value

Returns the parameter_value associated with the model.

columnCount(*parent=QModelIndex()*)

Returns the number of columns which is two.

data(*index*, *role=Qt.DisplayRole*)

Returns the data at index for given role.

headerData(*section*, *orientation=Qt.Horizontal*, *role=Qt.DisplayRole*)

Returns a header.

is_expanse_row(*row*)

Returns True if row is the expanse row.

Parameters
row (*int*) – a row

Returns
True if row is the expanse row, False otherwise

Return type
bool

reset(*value*)

Resets the model.

rowCount(*parent=QModelIndex()*)

Returns the number of rows.

setHeaderData(*section*, *orientation*, *value*, *role=Qt.EditRole*)

spinetoolbox.mvcmodels.map_model

A model for maps, used by the parameter_value editors.

authors

A. Soininen (VTT)

date

11.2.2020

Module Contents

Classes

MapModel	A model for Map type parameter values.

Functions

_rows_to_dict(rows)	Turns table into nested dictionaries.
_reconstruct_map(tree)	Constructs a `Map` from a nested dictionary.
_data_length(row)	Counts the number of non-empty elements at the beginning of row.
_gather_index_names(map_value)	Collects index names from Map.
_apply_index_names(map_value, index_names)	Applies index names to Map.
_numpy_string_to_python_strings(rows)	Converts instances of **numpy.str_** to regular Python strings.

Attributes

empty	Sentinel for empty cells.

`spinetoolbox.mvcmodels.map_model.`**`empty`**

Sentinel for empty cells.

class `spinetoolbox.mvcmodels.map_model.`**`MapModel`**(*map_value*, *parent*)

Bases: `PySide2.QtCore.QAbstractTableModel`

A model for Map type parameter values.

This model represents the Map as a 2D table. Each row consists of one or more index columns and a value column. The last columns of a row are padded with Nones.

Example

```
Map {
    "A": 1.0
    "B": Map {"a": -1.0}
    "C": 3.0
}
```

The table corresponding to the above map:

"A"	1.0	None
"B"	"a"	-1.0
"C"	3.0	None

Parameters

- **map_value** (*Map*) – a map
- **parent** (*QObject*) – parent object

append_column()

Appends a new column to the right.

clear(*indexes*)

Clears table cells.

Parameters
indexes (*list of QModelIndex*) – indexes to clear

columnCount(*index=QModelIndex()*)

Returns the number of columns in this model.

convert_leaf_maps()

data(*index*, *role=Qt.DisplayRole*)

Returns the data associated with the given role.

flags(*index*)

Returns flags at index.

headerData(*section*, *orientation*, *role=Qt.DisplayRole*)

Returns row numbers for vertical headers and column titles for horizontal ones.

insertColumns(*column*, *count*, *parent=QModelIndex()*)

Inserts new columns into the map.

Parameters

- **column** (*int*) – column index where to insert
- **count** (*int*) – number of new columns
- **parent** (*QModelIndex*) – ignored

Returns
True if insertion was successful, False otherwise

Return type
bool

insertRows(*row*, *count*, *parent=QModelIndex()*)

Inserts new rows into the map.

Parameters

- **row** (*int*) – an index where to insert the new data
- **count** (*int*) – number of rows to insert
- **parent** (*QModelIndex*) – an index to a parent model

Returns
True if the operation was successful

Return type
bool

is_leaf_value(*index*)

Checks if given model index contains a leaf value.

Parameters
index (*QModelIndex*) – index to check

Returns
True if index points to leaf value, False otherwise

Return type
bool

_is_in_expanse(*row*, *column*)

Returns True, if given row and column is in the right or bottom 'expanding' zone.

Parameters

- **row** (*int*) – row index
- **column** (*int*) – column index

Returns
True if the cell is in the expanse, False otherwise

Return type
bool

is_expanse_column(*column*)

Returns True if given column is the expanse column.

Parameters
column (*int*) – column

Returns
True if column is expanse column, False otherwise

Return type
bool

is_expanse_row(*row*)

Returns True if given row is the expanse row.

Parameters
row (*int*) – row

Returns
True if row is the expanse row, False otherwise

Return type
bool

`removeColumns`(*column*, *count*, *parent=QModelIndex()*)

Removes columns from the map.

Parameters

- **column** (*int*) – first column to remove
- **count** (*int*) – number of columns to remove
- **parent** (*QModelIndex*) – an index to a parent model

Returns
True if the operation was successful

`removeRows`(*row*, *count*, *parent=QModelIndex()*)

Removes rows from the map.

Parameters

- **row** (*int*) – first row to remove
- **count** (*int*) – number of rows to remove
- **parent** (*QModelIndex*) – an index to a parent model

Returns
True if the operation was successful

`reset`(*map_value*)

Resets the model to given map_value.

`rowCount`(*parent=QModelIndex()*)

Returns the number of rows.

`set_box`(*top_left*, *bottom_right*, *data*)

Sets data for several indexes at once.

Parameters

- **top_left** (*QModelIndex*) – a sequence of model indexes
- **bottom_right** (*QModelIndex*) – a sequence of values corresponding to the indexes
- **data** (*list of list*) – box of data

`setData`(*index*, *value*, *role=Qt.EditRole*)

Sets data in the map.

Parameters

- **index** (*QModelIndex*) – an index to the model
- **value** (*object*) – JSON representation of the value
- **role** (*int*) – a role

Returns
True if the operation was successful

Return type
bool

setHeaderData(*section*, *orientation*, *value*, *role=Qt.EditRole*)

trim_columns()

Removes empty columns from the right.

value()

Returns the Map.

index_name(*index*)

spinetoolbox.mvcmodels.map_model.**_rows_to_dict**(*rows*)

Turns table into nested dictionaries.

Parameters
rows (*list*) – a list of row data

Returns
a nested dictionary

Return type
dict

spinetoolbox.mvcmodels.map_model.**_reconstruct_map**(*tree*)

Constructs a `Map` from a nested dictionary.

Parameters
tree (*dict*) – a nested dictionary

Returns
reconstructed Map

Return type
Map

spinetoolbox.mvcmodels.map_model.**_data_length**(*row*)

Counts the number of non-empty elements at the beginning of row.

Parameters
row (*list*) – a row of data

Returns
data length

Return type
int

spinetoolbox.mvcmodels.map_model.**_gather_index_names**(*map_value*)

Collects index names from Map.

Returns only the 'first' index name for nested maps at the same depth.

Parameters
map_value (*Map*) – map to investigate

Returns
index names

Return type
list of str

`spinetoolbox.mvcmodels.map_model.`**`_apply_index_names`**(*map_value*, *index_names*)

Applies index names to Map.

Parameters

- **map_value** (*Map*) – target Map
- **index_names** (*list of str*) – index names

`spinetoolbox.mvcmodels.map_model.`**`_numpy_string_to_python_strings`**(*rows*)

Converts instances of **numpy.str_** to regular Python strings.

Parameters
rows (*list of list*) – table rows

Returns
converted rows

Return type
list of list

spinetoolbox.mvcmodels.minimal_table_model

Contains a minimal table model.

authors

M. Marin (KTH)

date
20.5.2018

Module Contents

Classes

MinimalTableModel	Table model for outlining simple tabular data.

class `spinetoolbox.mvcmodels.minimal_table_model.`**`MinimalTableModel`**(*parent=None*, *header=None*, *lazy=True*)

Bases: `PySide2.QtCore.QAbstractTableModel`

Table model for outlining simple tabular data.

Parameters

- **parent** (*QObject, optional*) – the parent object
- **header** (*list of str*) – header labels
- **lazy** (*boolean*) – if True, fetches data lazily

clear()

Clear all data in model.

flags(*index*)

Return index flags.

canFetchMore(*parent*)

Return True if the model hasn't been fetched.

fetchMore(*parent*)

Fetch data and use it to reset the model.

rowCount(*parent=QModelIndex()*)

Number of rows in the model.

columnCount(*parent=QModelIndex()*)

Number of columns in the model.

headerData(*section*, *orientation=Qt.Horizontal*, *role=Qt.DisplayRole*)

Returns headers.

set_horizontal_header_labels(*labels*)

Set horizontal header labels.

insert_horizontal_header_labels(*section*, *labels*)

Insert horizontal header labels at the given section.

horizontal_header_labels()

setHeaderData(*section*, *orientation*, *value*, *role=Qt.EditRole*)

Sets the data for the given role and section in the header with the specified orientation to the value supplied.

data(*index*, *role=Qt.DisplayRole*)

Returns the data stored under the given role for the item referred to by the index.

Parameters

- **index** (*QModelIndex*) – Index of item
- **role** (*int*) – Data role

Returns

Item data for given role.

row_data(*row*, *role=Qt.DisplayRole*)

Returns the data stored under the given role for the given row.

Parameters

- **row** (*int*) – Item row
- **role** (*int*) – Data role

Returns

Row data for given role.

setData(*index*, *value*, *role=Qt.EditRole*)

Set data in model.

batch_set_data(*indexes*, *data*)

Batch set data for indexes.

Parameters

- **indexes** (*Iterable of QModelIndex*) – model indexes
- **data** (*Iterable*) – data at each index

Returns
True if data was set successfully, False otherwise

Return type
boolean

`insertRows`(*row*, *count*, *parent=QModelIndex()*)

Inserts count rows into the model before the given row. Items in the new row will be children of the item represented by the parent model index.

Parameters

- **row** (*int*) – Row number where new rows are inserted
- **count** (*int*) – Number of inserted rows
- **parent** (*QModelIndex*) – Parent index

Returns
True if rows were inserted successfully, False otherwise

`insertColumns`(*column*, *count*, *parent=QModelIndex()*)

Inserts count columns into the model before the given column. Items in the new column will be children of the item represented by the parent model index.

Parameters

- **column** (*int*) – Column number where new columns are inserted
- **count** (*int*) – Number of inserted columns
- **parent** (*QModelIndex*) – Parent index

Returns
True if columns were inserted successfully, False otherwise

`removeRows`(*row*, *count*, *parent=QModelIndex()*)

Removes count rows starting with the given row under parent.

Parameters

- **row** (*int*) – Row number where to start removing rows
- **count** (*int*) – Number of removed rows
- **parent** (*QModelIndex*) – Parent index

Returns
True if rows were removed successfully, False otherwise

`removeColumns`(*column*, *count*, *parent=QModelIndex()*)

Removes count columns starting with the given column under parent.

Parameters

- **column** (*int*) – Column number where to start removing columns
- **count** (*int*) – Number of removed columns
- **parent** (*QModelIndex*) – Parent index

Returns
True if columns were removed successfully, False otherwise

`reset_model`(*main_data=None*)

Reset model.

spinetoolbox.mvcmodels.minimal_tree_model

Models to represent items in a tree.

authors

P. Vennström (VTT), M. Marin (KTH)

date

11.3.2019

Module Contents

Classes

TreeItem	A tree item that can fetch its children.
MinimalTreeModel	Base class for all tree models.

class spinetoolbox.mvcmodels.minimal_tree_model.**TreeItem**(*model=None*)

A tree item that can fetch its children.

Parameters

model (*MinimalTreeModel, optional*) – The model where the item belongs.

property model

property child_item_class

Returns the type of child items. Reimplement in subclasses to return something more meaningful.

property children

property parent_item

property display_data

property edit_data

has_children()

Returns whether or not this item has or could have children.

child(*row*)

Returns the child at given row or None if out of bounds.

last_child()

Returns the last child.

child_count()

Returns the number of children.

child_number()

Returns the rank of this item within its parent or -1 if it's an orphan.

find_children(*cond=lambda child: ...*)

Returns children that meet condition expressed as a lambda function.

find_child(*cond=lambda child: ...*)

Returns first child that meet condition expressed as a lambda function or None.

next_sibling()

Returns the next sibling or None if it's the last.

previous_sibling()

Returns the previous sibling or None if it's the first.

index()

finalize()

_do_finalize()

Do some final initialization after setting the parent.

insert_children(*position*, *children*)

Insert new children at given position. Returns a boolean depending on how it went.

Parameters

- **position** (*int*) – insert new items here
- **children** (*list of* TreeItem) – insert items from this iterable

append_children(*children*)

Append children at the end.

remove_children(*position*, *count*)

Removes count children starting from the given position.

Parameters

- **position** (*int*) – position of the first child to remove
- **count** (*int*) – number of children to remove

Returns

True if operation was successful, False otherwise

Return type

bool

clear_children()

Clear children list.

flags(*column*)

Enables the item and makes it selectable.

data(*column*, *role=Qt.DisplayRole*)

Returns data for given column and role.

can_fetch_more()

Returns whether or not this item can fetch more.

fetch_more()

Fetches more children.

abstract set_data(*column*, *value*, *role*)

Sets data for this item.

Parameters

- **column** (*int*) – column index
- **value** (*object*) – a new value
- **role** (*int*) – role of the new value

Returns
True if data was set successfully, False otherwise

Return type
bool

class spinetoolbox.mvcmodels.minimal_tree_model.**MinimalTreeModel**(*parent*)

Bases: PySide2.QtCore.QAbstractItemModel

Base class for all tree models.

Init class.

Parameters
parent (SpineDBEditor) –

visit_all(*index=QModelIndex()*, *view=None*)

Iterates all items in the model including and below the given index. Iterative implementation so we don't need to worry about Python recursion limits.

Parameters

- **index** (*QModelIndex*) – an index to start. If not given, we start at the root
- **view** (*QTreeView*) – a tree view. If given, we only yield items that are visible to that view. So for example, if a tree item is not expanded then we don't yield its children.

Yields
TreeItem

item_from_index(*index*)

Return the item corresponding to the given index.

index_from_item(*item*)

Return a model index corresponding to the given item.

index(*row*, *column*, *parent=QModelIndex()*)

Returns the index of the item in the model specified by the given row, column and parent index.

parent(*index*)

Returns the parent of the model item with the given index.

columnCount(*parent=QModelIndex()*)

rowCount(*parent=QModelIndex()*)

data(*index*, *role=Qt.DisplayRole*)

Returns the data stored under the given role for the index.

setData(*index*, *value*, *role=Qt.EditRole*)

Sets data for given index and role. Returns True if successful; otherwise returns False.

flags(*index*)

Returns the item flags for the given index.

hasChildren(*parent*)

canFetchMore(*parent*)

fetchMore(*parent*)

spinetoolbox.mvcmodels.project_item_model

Contains a class for storing project items.

authors

P. Savolainen (VTT)

date

23.1.2018

Module Contents

Classes

ProjectItemModel	Class to store project tree items and ultimately project items in a tree structure.

class spinetoolbox.mvcmodels.project_item_model.**ProjectItemModel**(*root*, *parent=None*)

Bases: PySide2.QtCore.QAbstractItemModel

Class to store project tree items and ultimately project items in a tree structure.

Parameters

- **root** (RootProjectTreeItem) – Root item for the project item tree
- **parent** (*QObject*) – parent object

root()

Returns the root item.

connect_to_project(*project*)

Connects the model to a project.

Parameters

project (SpineToolboxProject) – project to connect to

_add_leaf_item(*name*)

Adds a leaf item to the model

Parameters

name (*str*) – project item's name

_remove_leaf_item(*name*)

Removes a leaf item from the model.

Parameters

name (*str*) – project item's name

_rename_item(*old_name*, *new_name*)

Renames a leaf item.

Parameters

- **old_name** (*str*) – item's old name
- **new_name** (*str*) – item's new name

rowCount(*parent=QModelIndex()*)

Reimplemented rowCount method.

Parameters
parent (*QModelIndex*) – Index of parent item whose children are counted.

Returns
Number of children of given parent

Return type
int

columnCount(*parent=QModelIndex()*)

Returns model column count which is always 1.

flags(*index*)

Returns flags for the item at given index

Parameters
index (*QModelIndex*) – Flags of item at this index.

parent(*index=QModelIndex()*)

Returns index of the parent of given index.

Parameters
index (*QModelIndex*) – Index of item whose parent is returned

Returns
Index of parent item

Return type
QModelIndex

index(*row*, *column*, *parent=QModelIndex()*)

Returns index of item with given row, column, and parent.

Parameters

- **row** (*int*) – Item row
- **column** (*int*) – Item column
- **parent** (*QModelIndex*) – Parent item index

Returns
Item index

Return type
QModelIndex

data(*index*, *role=None*)

Returns data in the given index according to requested role.

Parameters

- **index** (*QModelIndex*) – Index to query
- **role** (*int*) – Role to return

Returns
Data depending on role.

Return type
object

item(*index*)

Returns item at given index.

Parameters
index (*QModelIndex*) – Index of item

Returns

Item at given index or root project
item if index is not valid

Return type
RootProjectTreeItem, *CategoryProjectTreeItem* or *LeafProjectTreeItem*

find_category(*category_name*)

Returns the index of the given category name.

Parameters
category_name (*str*) – Name of category item to find

Returns
index of a category item or None if it was not found

Return type
QModelIndex

find_item(*name*)

Returns the QModelIndex of the leaf item with the given name

Parameters
name (*str*) – The searched project item (long) name

Returns
Index of a project item with the given name or None if not found

Return type
QModelIndex

get_item(*name*)

Returns leaf item with given name or None if it doesn't exist.

Parameters
name (*str*) – Project item name

Returns
LeafProjectTreeItem, NoneType

category_of_item(*name*)

Returns the category item of the category that contains project item with given name

Parameters
name (*str*) – Project item name

Returns
category item or None if the category was not found

Return type
CategoryProjectTreeItem

insert_item(*item*, *parent=QModelIndex()*)

Adds a new item to model. Fails if given parent is not a category item nor a leaf item. New item is inserted as the last item of its branch.

Parameters

- **item** (CategoryProjectTreeItem *or* LeafProjectTreeItem) – Project item to add to model
- **parent** (*QModelIndex*) – Parent project item

Returns
True if successful, False otherwise

Return type
bool

remove_item(*item*, *parent=QModelIndex()*)

Removes item from project.

Parameters

- **item** (BaseProjectTreeItem) – Item to remove
- **parent** (*QModelIndex*) – Parent of item that is to be removed

Returns
True if item removed successfully, False if item removing failed

Return type
bool

items(*category_name=None*)

Returns a list of leaf items in model according to category name. If no category name given, returns all leaf items in a list.

Parameters
category_name (*str*) – Item category. Data Connections, Data Stores, Importers, Exporters, Tools or Views permitted.

Returns
obj:'list' of :obj:'LeafProjectTreeItem': Depending on category_name argument, returns all items or only items according to category. An empty list is returned if there are no items in the given category or if an unknown category name was given.

n_items()

Returns the number of all items in the model excluding category items and root.

Returns
Number of items

Return type
int

item_names()

Returns all leaf item names in a list.

Returns
'list' of obj:'str': Item names

Return type
obj

`items_per_category()`

Returns a dict mapping category indexes to a list of items in that category.

Returns
dict(QModelIndex,list(LeafProjectTreeItem))

`leaf_indexes()`

Yields leaf indexes.

`remove_leaves()`

spinetoolbox.mvcmodels.project_item_specification_models

Contains a class for storing Tool specifications.

authors

P. Savolainen (VTT)

date
23.1.2018

Module Contents

Classes

ProjectItemSpecificationModel	Class to store specs that are available in a project e.g. GAMS or Julia models.
FilteredSpecificationModel	

class spinetoolbox.mvcmodels.project_item_specification_models.**ProjectItemSpecificationModel**(*icons*)

Bases: `PySide2.QtCore.QAbstractListModel`

Class to store specs that are available in a project e.g. GAMS or Julia models.

`specification_replaced`

`add_specification`(*name*)

Adds a specification to the model.

Parameters
name (*str*) – specification's name

`remove_specification`(*name*)

Removes a specification from the model

Parameters
name (*str*) – specification's name

replace_specification(*old_name*, *new_name*)

Replaces a specification.

Parameters

- **old_name** (*str*) – previous name
- **new_name** (*str*) – new name

connect_to_project(*project*)

Connects the model to a project.

Parameters

project (SpineToolboxProject) – project to connect to

clear()

rowCount(*parent=None*)

Returns the number of specs in the model.

Parameters

parent (*QModelIndex*) – Not used (because this is a list)

Returns

Number of rows (available specs) in the model

data(*index*, *role=None*)

Must be reimplemented when subclassing.

Parameters

- **index** (*QModelIndex*) – Requested index
- **role** (*int*) – Data role

Returns

Data according to requested role

flags(*index*)

Returns enabled flags for the given index.

Parameters

index (*QModelIndex*) – Index of spec

insertRow(*spec_name*, *row=None*, *parent=QModelIndex()*)

Insert row (specification) into model.

Parameters

- **spec_name** (*str*) – name of spec added to the model
- **row** (*int, optional*) – Row to insert spec to
- **parent** (*QModelIndex*) – Parent of child (not used)

Returns

Void

removeRow(*row*, *parent=QModelIndex()*)

Remove row (spec) from model.

Parameters

- **row** (*int*) – Row to remove the spec from

- **parent** (*QModelIndex*) – Parent of spec on row (not used)

Returns
Boolean variable

specification(*row*)

Returns spec on given row.

Parameters
row (*int*) – Row of spec specification

Returns
ProjectItemSpecification from specification list or None if given row is zero

specification_row(*name*)

Returns the row on which the given specification is located or -1 if it is not found.

specification_index(*name*)

Returns the QModelIndex on which a specification with the given name is located or invalid index if it is not found.

class spinetoolbox.mvcmodels.project_item_specification_models.**FilteredSpecificationModel**(*item_type*)

Bases: PySide2.QtCore.QSortFilterProxyModel

filterAcceptsRow(*source_row*, *source_parent*)

get_mime_data_text(*index*)

specifications()

Yields all specs.

specification(*row*)

spinetoolbox.mvcmodels.project_tree_item

Project Tree items.

authors

A. Soininen (VTT)

date
17.1.2020

Module Contents

Classes

BaseProjectTreeItem	Base class for all project tree items.
RootProjectTreeItem	Class for the root project tree item.
CategoryProjectTreeItem	Class for category project tree items.
LeafProjectTreeItem	Class for leaf items in the project item tree.

class spinetoolbox.mvcmodels.project_tree_item.**BaseProjectTreeItem**(*name*, *description*)

Bases: *spinetoolbox.metaobject.MetaObject*

Base class for all project tree items.

Parameters

- **name** (*str*) – Object name
- **description** (*str*) – Object description

flags()

Returns the item flags.

parent()

Returns parent project tree item.

child_count()

Returns the number of child project tree items.

children()

Returns the children of this project tree item.

child(*row*)

Returns child BaseProjectTreeItem on given row.

Parameters
row (*int*) – Row of child to return

Returns
item on given row or None if it does not exist

Return type
BaseProjectTreeItem

row()

Returns the row on which this item is located.

abstract add_child(*child_item*)

Base method that shall be overridden in subclasses.

remove_child(*row*)

Remove the child of this BaseProjectTreeItem from given row. Do not call this method directly. This method is called by ProjectItemTreeModel when items are removed.

Parameters
row (*int*) – Row of child to remove

Returns
True if operation succeeded, False otherwise

Return type
bool

abstract custom_context_menu(*toolbox*)

Returns the context menu for this item. Implement in subclasses as needed.

Parameters
toolbox (*QWidget*) – The widget that is controlling the menu

Returns
context menu

Return type
QMenu

class spinetoolbox.mvcmodels.project_tree_item.**RootProjectTreeItem**

Bases: *BaseProjectTreeItem*

Class for the root project tree item.

Parameters

- **name** (*str*) – Object name
- **description** (*str*) – Object description

add_child(*child_item*)

Adds given category item as the child of this root project tree item. New item is added as the last item.

Parameters
child_item (CategoryProjectTreeItem) – Item to add

Returns
True for success, False otherwise

abstract custom_context_menu(*toolbox*)

See base class.

class spinetoolbox.mvcmodels.project_tree_item.**CategoryProjectTreeItem**(*name*, *description*)

Bases: *BaseProjectTreeItem*

Class for category project tree items.

Parameters

- **name** (*str*) – Object name
- **description** (*str*) – Object description

flags()

Returns the item flags.

add_child(*child_item*)

Adds given project tree item as the child of this category item. New item is added as the last item.

Parameters
child_item (*LeafProjectTreeTreeItem*) – Item to add

Returns
True for success, False otherwise

custom_context_menu(*toolbox*)

Returns the context menu for this item.

Parameters
toolbox (ToolboxUI) – Toolbox main window

Returns
context menu

Return type
QMenu

class spinetoolbox.mvcmodels.project_tree_item.**LeafProjectTreeItem**(*project_item*)

Bases: *BaseProjectTreeItem*

Class for leaf items in the project item tree.

Parameters
project_item (ProjectItem) – the real project item this item represents

property project_item

the project item linked to this leaf

abstract add_child(*child_item*)

See base class.

flags()

Returns the item flags.

custom_context_menu(*toolbox*)

Returns the context menu for this item.

Parameters
toolbox (ToolboxUI) – Toolbox main window

Returns
context menu

Return type
QMenu

spinetoolbox.mvcmodels.resource_filter_model

Contains ResourceFilterModel.

author

M. Marin (KTH)

date
26.11.2020

Module Contents

Classes

ResourceFilterModel	**param connection** connection whose resources to model

class spinetoolbox.mvcmodels.resource_filter_model.**ResourceFilterModel**(*connection*, *undo_stack*, *logger*)

Bases: PySide2.QtGui.QStandardItemModel

Parameters

- **connection** (LoggingConnection) – connection whose resources to model

- **undo_stack** (*QUndoStack*) – an undo stack
- **logger** (*LoggerInterface*) – a logger

property connection

tree_built

_SELECT_ALL = Select all

_FILTER_TYPES

_FILTER_TYPE_TO_TEXT

build_tree()

Rebuilds model's contents.

fetch_filters()

setData(*index*, *value*, *role=Qt.EditRole*)

_change_filter_checked_state(*index*, *is_on*)

Changes the online status of the filter item at index.

Parameters

- **index** (*QModelIndex*) – item's index
- **is_on** (*bool*) – True if filter are turned online, False otherwise

set_online(*resource*, *filter_type*, *online*)

Sets the given filters online or offline.

Parameters

- **resource** (*str*) – Resource label
- **filter_type** (*str*) – Either SCENARIO_FILTER_TYPE or TOOL_FILTER_TYPE, for now.
- **online** (*dict*) – mapping from scenario/tool id to online flag

_find_filter_type_item(*resource*, *filter_type*)

Searches for filter type item.

Parameters

- **resource** (*str*) – resource label
- **filter_type** (*str*) – filter type identifier

Returns
filter type item or None if not found

Return type
QStandardItem

_set_all_selected_item(*resource*, *filter_type_item*, *emit_data_changed=False*)

Updates 'Select All' item's checked state.

Parameters

- **resource** (*str*) – resource label
- **filter_type_item** (*QStandardItem*) – filter type item

- **emit_data_changed** (*bool*) – if True, emit dataChanged signal if the state was updated

spinetoolbox.mvcmodels.shared

Contains stuff that is used by more than one model

author

M. Marin (KTH)

date

23.3.2020

Module Contents

spinetoolbox.mvcmodels.shared.**PARSED_ROLE**

spinetoolbox.mvcmodels.shared.**DB_MAP_ROLE**

spinetoolbox.mvcmodels.time_pattern_model

A model for time patterns, used by the parameter_value editors.

authors

A. Soininen (VTT)

date

4.7.2019

Module Contents

Classes

TimePatternModel	A model for time pattern type parameter values.

class spinetoolbox.mvcmodels.time_pattern_model.**TimePatternModel**(*value*, *parent*)

Bases: *spinetoolbox.mvcmodels.indexed_value_table_model.IndexedValueTableModel*

A model for time pattern type parameter values.

Parameters

- **value** (*IndexedValue*) – a parameter_value
- **parent** (*QObject*) – parent object

flags(*index*)

Returns flags at index.

insertRows(*row*, *count*, *parent=QModelIndex()*)

Inserts new time period - value pairs into the pattern.

New time periods are initialized to empty strings and the corresponding values to zeros.

Parameters

- **row** (*int*) – an index where to insert the new data
- **count** (*int*) – number of time period - value pairs to insert
- **parent** (*QModelIndex*) – an index to a parent model

Returns
True if the operation was successful

Return type
bool

removeRows(*row*, *count*, *parent=QModelIndex()*)

Removes time period - value pairs from the pattern.

Parameters

- **row** (*int*) – an index where to remove the data
- **count** (*int*) – number of time period - value pairs to remove
- **parent** (*QModelIndex*) – an index to a parent model

Returns
True if the operation was successful

Return type
bool

setData(*index*, *value*, *role=Qt.EditRole*)

Sets a time period or a value in the pattern.

Column index 0 corresponds to the time periods while 1 corresponds to the values.

Parameters

- **index** (*QModelIndex*) – an index to the model
- **value** (*str, float*) – a new time period or value
- **role** (*int*) – a role

Returns
True if the operation was successful

Return type
bool

batch_set_data(*indexes*, *values*)

Sets data for several indexes at once.

Parameters

- **indexes** (*Sequence*) – a sequence of model indexes
- **values** (*Sequence*) – a sequence of time periods/floats corresponding to the indexes

spinetoolbox.mvcmodels.time_series_model_fixed_resolution

A model for fixed resolution time series, used by the parameter_value editors.

authors

A. Soininen (VTT)

date

4.7.2019

Module Contents

Classes

TimeSeriesModelFixedResolution	A model for fixed resolution time series type parameter values.

class spinetoolbox.mvcmodels.time_series_model_fixed_resolution.**TimeSeriesModelFixedResolution**(*series*, *parent*)

Bases: *spinetoolbox.mvcmodels.indexed_value_table_model.IndexedValueTableModel*

A model for fixed resolution time series type parameter values.

Parameters

- **series** (*TimeSeriesFixedResolution*) – a time series
- **parent** (*QObject*) – parent object

property indexes

Returns the time stamps as an array.

property values

Returns the values of the time series as an array.

flags(*index*)

Returns flags at index.

insertRows(*row*, *count*, *parent=QModelIndex()*)

Inserts new values to the series.

The new values are set to zero. Start time or resolution are left unchanged.

Parameters

- **row** (*int*) – a numeric index to the first stamp/value to insert
- **count** (*int*) – number of stamps/values to insert
- **parent** (*QModelIndex*) – index to a parent model

Returns

True if the operation was successful

removeRows(*row*, *count*, *parent=QModelIndex()*)

Removes values from the series.

Parameters

- **row** (*int*) – a numeric index to the series where to begin removing
- **count** (*int*) – how many stamps/values to remove
- **parent** (*QModelIndex*) – an index to the parent model

Returns
True if the operation was successful.

reset(*value*)

Resets the model with new time series data.

setData(*index*, *value*, *role=Qt.EditRole*)

Sets a given value in the series.

Column index 1 refers to values. Note it does not make sense to set the time stamps in fixed resolution series.

Parameters

- **index** (*QModelIndex*) – an index to the model
- **value** (*numpy.datetime64, float*) – a new stamp or value
- **role** (*int*) – a role

Returns
True if the operation was successful

batch_set_data(*indexes*, *values*)

Sets data for several indexes at once.

Only the values of the series are modified as the time stamps are immutable.

Parameters

- **indexes** (*Sequence*) – a sequence of model indexes
- **values** (*Sequence*) – a sequence of floats corresponding to the indexes

set_ignore_year(*ignore_year*)

Sets the ignore_year option of the time series.

set_repeat(*repeat*)

Sets the repeat option of the time series.

set_resolution(*resolution*)

Sets the resolution.

set_start(*start*)

Sets the start datetime.

spinetoolbox.mvcmodels.time_series_model_variable_resolution

A model for variable resolution time series, used by the parameter_value editors.

authors

A. Soininen (VTT)

date

5.7.2019

Module Contents

Classes

TimeSeriesModelVariableResolution	A model for variable resolution time series type parameter values.

class spinetoolbox.mvcmodels.time_series_model_variable_resolution.**TimeSeriesModelVariableResolution**(*val*
pa
ent

Bases: *spinetoolbox.mvcmodels.indexed_value_table_model.IndexedValueTableModel*

A model for variable resolution time series type parameter values.

Parameters

- **value** (*IndexedValue*) – a parameter_value
- **parent** (*QObject*) – parent object

property indexes

Returns the time stamps as an array.

property values

Returns the values of the time series as an array.

flags(*index*)

Returns the flags for given model index.

insertRows(*row*, *count*, *parent=QModelIndex()*)

Inserts new time stamps and values to the series.

When inserting in the middle of the series the new time stamps are distributed evenly among the time span between the two time stamps around the insertion point. When inserting at the beginning or at the end of the series the duration between the new time stamps is set equal to the first/last duration in the original series.

The new values are set to zero.

Parameters

- **row** (*int*) – a numeric index to the first stamp/value to insert
- **count** (*int*) – number of stamps/values to insert
- **parent** (*QModelIndex*) – index to a parent model

Returns
True if the insertion was successful

Return type
bool

`removeRows`(*row*, *count*, *parent=QModelIndex()*)

Removes time stamps/values from the series.

Parameters

- **row** (*int*) – a numeric index to the series where to begin removing
- **count** (*int*) – how many stamps/values to remove
- **parent** (*QModelIndex*) – an index to the parent model

Returns
True if the operation was successful.

Return type
bool

`reset`(*value*)

Resets the model with new time series data.

`setData`(*index*, *value*, *role=Qt.EditRole*)

Sets a given time stamp or value in the series.

Column index 0 refers to time stamps while index 1 to values.

Parameters

- **index** (*QModelIndex*) – an index to the model
- **value** (*numpy.datetime64, float*) – a new stamp or value
- **role** (*int*) – a role

Returns
True if the operation was successful

Return type
bool

`batch_set_data`(*indexes*, *values*)

Sets data for several indexes at once.

Parameters

- **indexes** (*Sequence*) – a sequence of model indexes
- **values** (*Sequence*) – a sequence of datetimes/floats corresponding to the indexes

`set_ignore_year`(*ignore_year*)

Sets the ignore_year option of the time series.

`set_repeat`(*repeat*)

Sets the repeat option of the time series.

spinetoolbox.project_item

This subpackage contains base classes for project items.

authors

M. Marin (KTH)

date

8.10.2020

Submodules

spinetoolbox.project_item.logging_connection

Contains logging connection and jump classes.

Module Contents

Classes

HeadlessConnection	A project item connection that is compatible with headless mode.
LoggingConnection	A project item connection that is compatible with headless mode.
LoggingJump	

class spinetoolbox.project_item.logging_connection.**HeadlessConnection**(*source_name*, *source_position*, *destination_name*, *destination_position*, *options=None*, *disabled_filter_names=None*, *legacy_resource_filter_ids=None*)

Bases: spine_engine.project_item.connection.ResourceConvertingConnection

A project item connection that is compatible with headless mode.

property database_resources

Connection's database resources

set_filter_enabled(*resource_label*, *filter_type*, *filter_name*, *enabled*)

Enables or disables a filter.

Parameters

- **resource_label** (*str*) – database resource name
- **filter_type** (*str*) – filter type
- **filter_name** (*str*) – filter name
- **enabled** (*bool*) – True to enable the filter, False to disable it

_convert_legacy_resource_filter_ids_to_disabled_filter_names()

Converts legacy resource filter ids to disabled filter names.

This method should be called once after constructing the connection from potentially legacy dict using `from_dict()`.

static _constructor_args_from_dict(*connection_dict*)

See base class.

classmethod from_dict(*connection_dict*, ***kwargs*)

Deserializes a connection from dict.

Parameters

- **connection_dict** (*dict*) – serialized LoggingConnection
- ****kwargs** – additional keyword arguments to be forwarded to class constructor

receive_resources_from_source(*resources*)

See base class.

replace_resources_from_source(*old*, *new*)

Replaces existing resources by new ones.

Parameters

- **old** (*list of ProjectItemResource*) – old resources
- **new** (*list of ProjectItemResource*) – new resources

class spinetoolbox.project_item.logging_connection.**LoggingConnection**(**args*, *toolbox*, ***kwargs*)

Bases: *spinetoolbox.log_mixin.LogMixin*, *HeadlessConnection*

A project item connection that is compatible with headless mode.

property graphics_item

__hash__()

Return hash(self).

static item_type()

has_filters()

Returns True if connection has scenario or tool filters.

Returns
True if connection has filters, False otherwise

Return type
bool

_get_db_map(*url*)

_pop_unused_db_maps()

Removes unused database maps and unregisters from listening the DB manager.

_fetch_more_if_possible()

_receive_data_changed()

receive_scenarios_added(*_db_map_data*)

receive_scenarios_removed(*_db_map_data*)

receive_scenarios_updated(*_db_map_data*)

receive_tools_added(*_db_map_data*)

receive_tools_removed(*_db_map_data*)

receive_tools_updated(*_db_map_data*)

receive_session_committed(*db_maps*, *cookie*)

receive_session_rolled_back(*db_map*)

get_scenario_names(*url*)

get_tool_names(*url*)

may_have_filters()

Returns whether this connection may have filters.

Returns
True if it is possible for the connection to have filters, False otherwise

Return type
bool

may_have_write_index()

Returns whether this connection may have write index.

Returns
True if it is possible for the connection to have write index, False otherwise

Return type
bool

may_use_memory_db()

Returns whether this connection may use memory DB.

Returns
True if it is possible for the connection to use memory DB, False otherwise

Return type
bool

may_use_datapackage()

Returns whether this connection may use datapackage.

Returns
True if it is possible for the connection to use datapackage, False otherwise

Return type
bool

may_purge_before_writing()

Returns whether this connection may purge before writing.

Returns
True if it is possible for the connection to purge before writing, False otherwise

Return type
bool

disabled_filter_names(*resource_label*, *filter_type*)

Returns disabled filter names for given resource and filter type.

Parameters

- **resource_label** (*str*) – resource label
- **filter_type** (*str*) – filter type

Returns
names of disabled filters

Return type
set of str

set_online(*resource*, *filter_type*, *online*)

Sets the given filters online or offline.

Parameters

- **resource** (*str*) – Resource label
- **filter_type** (*str*) – Either SCENARIO_FILTER_TYPE or TOOL_FILTER_TYPE, for now.
- **online** (*dict*) – mapping from scenario/tool id to online flag

refresh_resource_filter_model()

Makes resource filter mode fetch filter data from database.

receive_resources_from_source(*resources*)

See base class.

replace_resources_from_source(*old*, *new*)

See base class.

set_connection_options(*options*)

Overwrites connections options.

Parameters
options (*dict*) – new options

_mask_unavailable_disabled_filters()

Cross-checks disabled filters with source databases.

Returns
disabled filter names containing only names that exist in source databases

Return type
dict

to_dict()

See base class.

tear_down()

Releases system resources held by the connection.

class spinetoolbox.project_item.logging_connection.**LoggingJump**(**args*, *toolbox=None*, ***kwargs*)

Bases: *spinetoolbox.log_mixin.LogMixin*, spine_engine.project_item.connection.Jump

property graphics_item

static item_type()

spinetoolbox.project_item.project_item

Contains base classes for project items and item factories.

authors

P. Savolainen (VTT)

date

4.10.2018

Module Contents

Classes

ProjectItem	Class for project items that are not category nor root.

class spinetoolbox.project_item.project_item.**ProjectItem**(*name*, *description*, *x*, *y*, *project*)

Bases: *spinetoolbox.log_mixin.LogMixin*, *spinetoolbox.metaobject.MetaObject*

Class for project items that are not category nor root. These items can be executed, refreshed, and so on.

x

horizontal position in the screen

Type

float

y

vertical position in the screen

Type

float

Parameters

- **name** (*str*) – item name
- **description** (*str*) – item description
- **x** (*float*) – horizontal position on the scene
- **y** (*float*) – vertical position on the scene
- **project** (SpineToolboxProject) – project item's project

property logger

abstract property executable_class

create_data_dir()

data_files()

Returns a list of files that are in the data directory.

abstract static item_type()

Item's type identifier string.

Returns
type string

Return type
str

abstract static item_category()

Item's category.

Returns
category name

Return type
str

make_signal_handler_dict()

Returns a dictionary of all shared signals and their handlers. This is to enable simpler connecting and disconnecting. Must be implemented in subclasses.

activate()

Restore selections and connect signals.

deactivate()

Save selections and disconnect signals.

restore_selections()

Restore selections into shared widgets when this project item is selected.

save_selections()

Save selections in shared widgets for this project item into instance variables.

_connect_signals()

Connect signals to handlers.

_disconnect_signals()

Disconnect signals from handlers and check for errors.

set_properties_ui(*properties_ui*)

Sets the properties tab widget for the item.

Note that this method expects the widget that is generated from the .ui files and initialized with the setupUi() method rather than the entire properties tab widget.

Parameters
properties_ui (*QWidget*) – item's properties UI

specification()

Returns the specification for this item.

undo_specification()

set_specification(*specification*)

Pushes a new SetItemSpecificationCommand to the toolbox' undo stack.

do_set_specification(*specification*)

Sets specification for this item. Removes specification if None given as argument.

Parameters
specification (*ProjectItemSpecification*) – specification of this item. None removes the specification.

set_icon(*icon*)

Sets the icon for the item.

Parameters
icon (ProjectItemIcon) – item's icon

get_icon()

Returns the graphics item representing this item in the scene.

_check_notifications()

Checks if exclamation icon notifications need to be set or cleared.

clear_notifications()

Clear all notifications from the exclamation icon.

add_notification(*text*)

Add a notification to the exclamation icon.

remove_notification(*text*)

set_rank(*rank*)

Set rank of this item for displaying in the design view.

handle_execution_successful(*execution_direction*, *engine_state*)

Performs item dependent actions after the execution item has finished successfully.

Parameters

- **execution_direction** (*str*) – "FORWARD" or "BACKWARD"
- **engine_state** – engine state after item's execution

resources_for_direct_successors()

Returns resources for direct successors.

These resources can include transient files that don't exist yet, or filename patterns. The default implementation returns an empty list.

Returns
a list of ProjectItemResources

Return type
list

resources_for_direct_predecessors()

Returns resources for direct predecessors.

These resources can include transient files that don't exist yet, or filename patterns. The default implementation returns an empty list.

Returns
a list of ProjectItemResources

Return type
list

_resources_to_predecessors_changed()

Notifies direct predecessors that item's resources have changed.

_resources_to_predecessors_replaced(*old*, *new*)

Notifies direct predecessors that item's resources have been replaced.

Parameters

- **old** (*list of ProjectItemResource*) – old resources
- **new** (*list of ProjectItemResource*) – new resources

upstream_resources_updated(*resources*)

Notifies item that resources from direct predecessors have changed.

Parameters
resources (*list of ProjectItemResource*) – new resources from upstream

replace_resources_from_upstream(*old*, *new*)

Replaces existing resources from direct predecessor by a new ones.

Parameters

- **old** (*list of ProjectItemResource*) – old resources
- **new** (*list of ProjectItemResource*) – new resources

_resources_to_successors_changed()

Notifies direct successors that item's resources have changed.

_resources_to_successors_replaced(*old*, *new*)

Notifies direct successors that one of item's resources has been replaced.

Parameters

- **old** (*list of ProjectItemResource*) – old resources
- **new** (*list of ProjectItemResource*) – new resources

downstream_resources_updated(*resources*)

Notifies item that resources from direct successors have changed.

Parameters
resources (*list of ProjectItemResource*) – new resources from downstream

replace_resources_from_downstream(*old*, *new*)

Replaces existing resources from direct successor by a new ones.

Parameters

- **old** (*list of ProjectItemResource*) – old resources
- **new** (*list of ProjectItemResource*) – new resources

invalidate_workflow(*edges*)

Notifies that this item's workflow is not acyclic.

Parameters
edges (*list*) – A list of edges that make the graph acyclic after removing them.

revalidate_workflow()

item_dict()

Returns a dictionary corresponding to this item.

Returns
serialized project item

Return type
dict

static item_dict_local_entries()

Returns entries or 'paths' in item dict that should be stored in project's local data directory.

Returns
local data item dict entries

Return type
list of tuple of str

static parse_item_dict(*item_dict*)

Reads the information needed to construct the base ProjectItem class from an item dict.

Parameters
item_dict (*dict*) – an item dict

Returns
item's name, description as well as x and y coordinates

Return type
tuple

copy_local_data(*item_dict*)

Copies local data linked to a duplicated project item.

Parameters
item_dict (*dict*) – serialized item

abstract static from_dict(*name*, *item_dict*, *toolbox*, *project*)

Deserialized an item from item dict.

Parameters

- **name** (*str*) – item's name
- **item_dict** (*dict*) – serialized item
- **toolbox** (ToolboxUI) – the main window
- **project** (SpineToolboxProject) – a project

Returns
deserialized item

Return type
ProjectItem

actions()

Item specific actions.

Returns
item's actions

Return type
list of QAction

rename(*new_name*, *rename_data_dir_message*)

Renames this item.

If the project item needs any additional steps in renaming, override this method in subclass. See e.g. rename() method in DataStore class.

Parameters

- **new_name** (*str*) – New name
- **rename_data_dir_message** (*str*) – Message to show when renaming item's data directory

Returns
True if item was renamed successfully, False otherwise

Return type
bool

open_directory(*checked=False*)

Open this item's data directory in file explorer.

tear_down()

Tears down this item. Called both before closing the app and when removing the item from the project. Implement in subclasses to eg close all QMainWindows opened by this item.

set_up()

Sets up this item. Called when adding the item to the project. Implement in subclasses to eg recreate attributes destroyed by tear_down.

update_name_label()

Updates the name label on the properties widget, used when selecting an item and renaming the selected one.

notify_destination(*source_item*)

Informs an item that it has become the destination of a connection between two items.

The default implementation logs a warning message. Subclasses should reimplement this if they need more specific behavior.

Parameters
source_item (ProjectItem) – connection source item

static upgrade_v1_to_v2(*item_name*, *item_dict*)

Upgrades item's dictionary from v1 to v2.

Subclasses should reimplement this method if there are changes between version 1 and version 2.

Parameters

- **item_name** (*str*) – item's name
- **item_dict** (*dict*) – Version 1 item dictionary

Returns
Version 2 item dictionary

Return type
dict

static upgrade_v2_to_v3(*item_name*, *item_dict*, *project_upgrader*)

Upgrades item's dictionary from v2 to v3.

Subclasses should reimplement this method if there are changes between version 2 and version 3.

Parameters

- **item_name** (*str*) – item's name
- **item_dict** (*dict*) – Version 2 item dictionary
- **project_upgrader** (ProjectUpgrader) – Project upgrader class instance

Returns
Version 3 item dictionary

Return type
dict

spinetoolbox.project_item.project_item_factory

Contains base classes for project items and item factories.

authors

P. Savolainen (VTT)

date
4.10.2018

Module Contents

Classes

ProjectItemFactory	Class for project item factories.

class spinetoolbox.project_item.project_item_factory.**ProjectItemFactory**

Class for project item factories.

abstract static item_class()

Returns the project item's class.

Returns
item's class

Return type
type

static is_deprecated()

Queries if item is deprecated.

Returns
True if item is deprecated, False otherwise

Return type
bool

abstract static icon()

Returns the icon resource path.

Returns
str

abstract static icon_color()

Returns the icon color.

Returns
icon's color

Return type
QColor

abstract static make_add_item_widget(*toolbox*, *x*, *y*, *specification*)

Returns an appropriate Add project item widget.

Parameters

- **toolbox** (ToolboxUI) – the main window
- **x** (*int*) – Icon coordinates
- **y** (*int*) – Icon coordinates
- **specification** (*ProjectItemSpecification*) – item's specification

Returns
QWidget

abstract static make_icon(*toolbox*)

Returns a ProjectItemIcon to use with given toolbox, for given project item.

Parameters
toolbox (ToolboxUI) –

Returns
item's icon

Return type
ProjectItemIcon

abstract static make_item(*name*, *item_dict*, *toolbox*, *project*)

Returns a project item constructed from the given `item_dict`.

Parameters

- **name** (*str*) – item's name
- **item_dict** (*dict*) – serialized project item
- **toolbox** (ToolboxUI) – Toolbox main window
- **project** (SpineToolboxProject) – the project the item belongs to

Returns
ProjectItem

abstract static make_properties_widget(*toolbox*)

Creates the item's properties tab widget.

Returns
item's properties tab widget

Return type
QWidget

abstract static make_specification_menu(*parent*, *index*)

Creates item specification's context menu.

Subclasses that do not support specifications can still raise `NotImplementedError`.

Parameters

- **parent** (*QWidget*) – menu's parent widget
- **index** (*QModelIndex*) – an index from specification model

Returns
specification's context menu

Return type
ItemSpecificationMenu

abstract static make_specification_editor(*toolbox*, *specification=None*, *item=None*, ***kwargs*)

Creates the item's specification widget.

Subclasses that do not support specifications can still raise `NotImplementedError`.

Parameters

- **toolbox** (*ToolboxUI*) – Toolbox main window
- **specification** (*ProjectItemSpecification, optional*) – a specification to show in the widget or None for a fresh start
- **item** (*ProjectItem, optional*) – a project item. If the specification is accepted, it is also set for this item
- ****kwargs** – parameters passed to the specification widget

Returns
item's specification widget

Return type
QWidget

static repair_specification(*toolbox*, *specification*)

Called right after a spec is added to the project. Finds if there's something wrong with the spec and proposes actions to fix it with help from toolbox.

Parameters

- **toolbox** (*ToolboxUI*) – Toolbox main window
- **specification** (*ProjectItemSpecification*) – a specification to check

spinetoolbox.project_item.specification_editor_window

Contains SpecificationEditorWindowBase and ChangeSpecPropertyCommand

author

M. Marin (KTH), P. Savolainen (VTT)

date

12.4.2018

Module Contents

Classes

ChangeSpecPropertyCommand	Command to set specification properties.
SpecificationEditorWindowBase	Base class for spec editors.
_SpecNameDescriptionToolbar	A QToolBar to let users set name and description for an Spec.

Functions

prompt_to_save_changes(parent, settings, save_callback)	Prompts to save changes.

class spinetoolbox.project_item.specification_editor_window.**ChangeSpecPropertyCommand**(*callback*, *new_value*, *old_value*, *cmd_name*)

Bases: PySide2.QtWidgets.QUndoCommand

Command to set specification properties.

Parameters

- **callback** (*function*) – Function to call to set the spec property.
- **new_value** (*any*) – new value
- **old_value** (*any*) – old value
- **cmd_name** (*str*) – command name

redo()

undo()

class spinetoolbox.project_item.specification_editor_window.**SpecificationEditorWindowBase**(*toolbox*, *specification=None*, *item=None*)

Bases: `PySide2.QtWidgets.QMainWindow`

Base class for spec editors.

Parameters

- **toolbox** (ToolboxUI) – QMainWindow instance
- **specification** (*ProjectItemSpecification, optional*) – If given, the form is prefilled with this specification
- **item** (ProjectItem, *optional*) – Sets the spec for this item if accepted

abstract property settings_group

Returns the settings group for this spec type.

Returns
str

property _duplicate_kwargs

abstract _make_ui()

Returns the ui object from Qt designer.

Returns
object

_restore_dock_widgets()

Restores dockWidgets to some default state. Called in the constructor, before restoring the ui from settings. Reimplement in subclasses if needed.

abstract _make_new_specification(*spec_name*)

Returns a ProjectItemSpecification from current form settings.

Parameters
spec_name (*str*) – Name of the spec

Returns
ProjectItemSpecification

show_error(*message*)

_show_status_bar_msg(*msg*)

_populate_main_menu()

_update_window_modified(*clean*)

_save()

Saves spec.

Returns
True if operation was successful, False otherwise

Return type
bool

_duplicate()

tear_down()

closeEvent(*event*)

class spinetoolbox.project_item.specification_editor_window.**_SpecNameDescriptionToolbar**(*parent*, *spec*, *undo_stack*)

Bases: PySide2.QtWidgets.QToolBar

A QToolBar to let users set name and description for an Spec.

Parameters

- **parent** (*QMainWindow*) – QMainWindow instance
- **spec** (*ProjectItemSpecification*) – specification that is being edited
- **undo_stack** (*QUndoStack*) – an undo stack

_make_main_menu()

_set_name()

_set_description()

do_set_name(*name*)

do_set_description(*description*)

name()

description()

spinetoolbox.project_item.specification_editor_window.**prompt_to_save_changes**(*parent*, *settings*, *save_callback*)

Prompts to save changes.

Parameters

- **parent** (*QWidget*) – Spec editor widget
- **settings** (*QSettings*) – Toolbox settings
- **save_callback** (*Callable*) – A function to call if the user chooses Save. It must return True or False depending on the outcome of the 'saving'.

Returns
False if the user chooses to cancel, in which case we don't close the form.

Return type
bool

spinetoolbox.server

Package for handling the client part of executing projects on Spine Engine Server.

authors

P. Pääkkönen (VTT), P. Savolainen (VTT)

date
02.09.2021

Submodules

spinetoolbox.server.engine_client

Client for exchanging messages between the toolbox and the Spine Engine Server. :author: P. Pääkkönen (VTT), P. Savolainen (VTT) :date: 02.09.2021

Module Contents

Classes

ClientSecurityModel	Generic enumeration.
EngineClient	**param host** IP address of the Spine Engine Server

class spinetoolbox.server.engine_client.**ClientSecurityModel**

Bases: enum.Enum

Generic enumeration.

Derive from this class to define new enumerations.

NONE = 0

STONEHOUSE = 1

class spinetoolbox.server.engine_client.**EngineClient**(*host*, *port*, *sec_model*, *sec_folder*, *ping=True*)

Parameters

- **host** (*str*) – IP address of the Spine Engine Server
- **port** (*int*) – Port of the client facing (frontend) socket on Spine Engine Server
- **sec_model** (ClientSecurityModel) – Client security scheme
- **sec_folder** (*str*) – Path to security file directory
- **ping** (*bool*) – Whether to check connectivity at instance creation

connect_pull_socket(*port*)

Connects a PULL socket for receiving engine execution events and files from server.

Parameters

port (*str*) – Port of the PUSH socket on server

rcv_next(*dealer_or_pull*)

Polls all sockets and returns a new reply based on given socket 'name'.

Parameters

dealer_or_pull (*str*) – "dealer" to wait reply from DEALER socket, "pull" to wait reply from PULL socket

_check_connectivity(*timeout*)

Pings server, waits for the response, and acts accordingly.

Parameters
timeout (*int*) – Time to wait for a response before giving up [ms]

Returns
void

Raises
RemoteEngineInitFailed if the server is not responding. –

set_start_time()

Sets a start time for an operation. Call get_elapsed_time() after an operation has finished to get the elapsed time string.

upload_project(*project_dir_name*, *fpath*)

Uploads the zipped project file to server. Project zip file must be ready and the server available before calling this method.

Parameters

- **project_dir_name** (*str*) – Project directory name
- **fpath** (*str*) – Absolute path to zipped project file.

Returns
Project execution job Id

Return type
str

start_execution(*engine_data*, *job_id*)

Sends the start execution request along with job Id and engine (dag) data to the server. Response message data contains the push/pull socket port if execution starts successfully.

Parameters

- **engine_data** (*str*) – Input for SpineEngine as JSON str. Includes most of project.json, settings, etc.
- **job_id** (*str*) – Project execution job Id on server

Returns
Response tuple (event_type, data). Event_type is "server_init_failed", "remote_execution_init_failed" or "remote_execution_started. data is an error message or the publish and push sockets ports concatenated with ':'.

Return type
tuple

stop_execution(*job_id*)

Sends a request to stop executing the DAG that is managed by this client.

Parameters
job_id (*str*) – Job Id on server to stop

download_files(*q*)

Pulls files from server until b'END' is received.

save_downloaded_file(*b_rel_path*, *file_data*)

Saves downloaded file to project directory.

Parameters

- **b_rel_path** (*bytes*) – Relative path (to project dir) where the file should be saved
- **file_data** (*bytes*) – File as bytes object

retrieve_project(*job_id*)

Retrieves a zipped project file from server.

Parameters
job_id (*str*) – Job Id for finding the project directory on server

Returns
Zipped project file

Return type
bytes

send_is_complete(*persistent_key*, *cmd*)

Sends a request to process is_complete(cmd) in persistent manager on server and returns the response.

send_issue_persistent_command(*persistent_key*, *cmd*)

Sends a request to process given command in persistent manager identified by given key. Yields the response string(s) as they arrive from server.

send_get_persistent_completions(*persistent_key*, *text*)

Requests completions to given text from persistent execution backend.

send_get_persistent_history_item(*persistent_key*, *text*, *prefix*, *backwards*)

Requests the former or latter history item from persistent execution backend.

send_restart_persistent(*persistent_key*)

Sends restart persistent cmd to persistent execution manager backend on server. Yields the messages resulting from this operation to persistent console client.

send_interrupt_persistent(*persistent_key*)

Sends interrupt persistent cmd to persistent execution manager backend on server.

send_request_to_persistent(*data*)

Sends given data containing persistent_key, command, cmd_to_persistent to Spine Engine Server to be processed by a persistent execution manager backend. Makes a request using REQ socket, parses the response into a ServerMessage, and returns the second part of the data field.

send_request_to_persistent_generator(*data*)

Pulls all messages from server, that were the result of sending given data to Spine Engine Server.

get_elapsed_time()

Returns the elapsed time between now and when self.start_time was set.

Returns
Time string with unit(s)

Return type
str

close()

Closes client sockets, context and thread.

spinetoolbox.spine_db_editor

This subpackage contains GUI files for the Spine db editor.

authors

M. Marin (KTH)

date

13.5.2020

Subpackages

spinetoolbox.spine_db_editor.mvcmodels

Modules in this package contain classes that represent Spine Toolbox's models (internal data structures) in the Model-View-Controller design pattern. The model classes define an interface that is used by views and delegates to access data in the application.

author

M. Marin (KTH)

date

23.5.2020

Submodules

spinetoolbox.spine_db_editor.mvcmodels.alternative_scenario_item

Classes to represent alternative and scenario items in a tree.

authors

P. Vennström (VTT)

date

17.6.2020

Module Contents

Classes

AlternativeRootItem	An alternative root item.
ScenarioRootItem	A scenario root item.
AlternativeLeafItem	An alternative leaf item.
ScenarioLeafItem	A scenario leaf item.
ScenarioActiveItem	A tree item that fetches their children as they are inserted.
ScenarioAlternativeRootItem	A scenario alternative root item.
ScenarioAlternativeLeafItem	A scenario alternative leaf item.

Attributes

_ALTERNATIVE_ICON
_SCENARIO_ICON

spinetoolbox.spine_db_editor.mvcmodels.alternative_scenario_item.**_ALTERNATIVE_ICON** =

spinetoolbox.spine_db_editor.mvcmodels.alternative_scenario_item.**_SCENARIO_ICON** =

class spinetoolbox.spine_db_editor.mvcmodels.alternative_scenario_item.**AlternativeRootItem**(**args*, ***kwargs*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.EmptyChildRootItem*

An alternative root item.

property item_type

property display_data

property icon_code

empty_child()

class spinetoolbox.spine_db_editor.mvcmodels.alternative_scenario_item.**ScenarioRootItem**(**args*, ***kwargs*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.EmptyChildRootItem*

A scenario root item.

property item_type

property display_data

property icon_code

empty_child()

class spinetoolbox.spine_db_editor.mvcmodels.alternative_scenario_item.**AlternativeLeafItem**(*identifier=None*

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.GrayIfLastMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.EditableMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.LeafItem*

An alternative leaf item.

Parameters
model (*MinimalTreeModel*, *optional*) – The model where the item belongs.

property item_type

property tool_tip

add_item_to_db(*db_item*)

update_item_in_db(*db_item*)

flags(*column*)

Makes items editable.

class spinetoolbox.spine_db_editor.mvcmodels.alternative_scenario_item.**ScenarioLeafItem**(*identifier=None*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.GrayIfLastMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.EditableMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.LeafItem*

A scenario leaf item.

Parameters

model (*MinimalTreeModel, optional*) – The model where the item belongs.

property item_type

property scenario_alternative_root_item

add_item_to_db(*db_item*)

update_item_in_db(*db_item*)

_do_finalize()

Do some final initialization after setting the parent.

handle_updated_in_db()

class spinetoolbox.spine_db_editor.mvcmodels.alternative_scenario_item.**ScenarioActiveItem**(*model=None*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.StandardTreeItem*

A tree item that fetches their children as they are inserted.

Parameters

model (*MinimalTreeModel, optional*) – The model where the item belongs.

property item_type

flags(*column*)

Enables the item and makes it selectable.

data(*column*, *role=Qt.DisplayRole*)

Returns data for given column and role.

set_data(*column*, *value*, *role=Qt.EditRole*)

Sets data for this item.

Parameters

- **column** (*int*) – column index
- **value** (*object*) – a new value
- **role** (*int*) – role of the new value

Returns

True if data was set successfully, False otherwise

Return type

bool

class spinetoolbox.spine_db_editor.mvcmodels.alternative_scenario_item.**ScenarioAlternativeRootItem**(**args*, ***kw*

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.EmptyChildRootItem*

A scenario alternative root item.

property item_type

property display_data

property tool_tip

property icon_code

property alternative_id_list

empty_child()

flags(*column*)

Enables the item and makes it selectable.

update_alternative_id_list()

class spinetoolbox.spine_db_editor.mvcmodels.alternative_scenario_item.**ScenarioAlternativeLeafItem**(*identi*

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.GrayIfLastMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.LeafItem*

A scenario alternative leaf item.

Parameters

model (*MinimalTreeModel, optional*) – The model where the item belongs.

property item_type

property tool_tip

property item_data

property alternative_id

_make_item_data()

abstract add_item_to_db(*db_item*)

abstract update_item_in_db(*db_item*)

flags(*column*)

Enables the item and makes it selectable.

set_data(*column*, *value*, *role=Qt.EditRole*)

Sets data for this item.

Parameters

- **column** (*int*) – column index
- **value** (*object*) – a new value
- **role** (*int*) – role of the new value

Returns

True if data was set successfully, False otherwise

Return type
bool

spinetoolbox.spine_db_editor.mvcmodels.alternative_scenario_model

Models to represent alternatives, scenarios and scenario alternatives in a tree.

authors

P. Vennström (VTT), M. Marin (KTH)

date
17.6.2020

Module Contents

Classes

AlternativeScenarioModel	A model to display alternatives and scenarios in a tree view.

class spinetoolbox.spine_db_editor.mvcmodels.alternative_scenario_model.**AlternativeScenarioModel**(*parent*, *db_mng*, **db_ma*

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_model_base.TreeModelBase*

A model to display alternatives and scenarios in a tree view.

Parameters

- **parent** (SpineDBEditor) –
- **db_mngr** (SpineDBManager) –
- ***db_maps** – DiffDatabaseMapping instances

static _make_db_item(*db_map*)

static _top_children()

_scenarios_per_root(*db_map_data*)

_alternatives_per_root(*db_map_data*)

add_alternatives(*db_map_data*)

add_scenarios(*db_map_data*)

update_alternatives(*db_map_data*)

update_scenarios(*db_map_data*)

remove_alternatives(*db_map_data*)

remove_scenarios(*db_map_data*)

supportedDropActions()

mimeData(*indexes*)

Builds a dict mapping db name to item type to a list of ids.

Returns
QMimeData

canDropMimeData(*data*, *drop_action*, *row*, *column*, *parent*)

dropMimeData(*data*, *drop_action*, *row*, *column*, *parent*)

spinetoolbox.spine_db_editor.mvcmodels.colors

Color constants for models.

authors

A. Soininen (VTT)

date
30.3.2022

Module Contents

spinetoolbox.spine_db_editor.mvcmodels.colors.**FIXED_FIELD_COLOR**

spinetoolbox.spine_db_editor.mvcmodels.compound_parameter_models

Compound models for object parameter definitions and values. These models concatenate several 'single' models and one 'empty' model.

authors

M. Marin (KTH)

date
28.6.2019

Module Contents

Classes

CompoundParameterModel	A model that concatenates several single parameter models
CompoundObjectParameterMixin	Implements the interface for populating and filtering a compound object parameter model.
CompoundRelationshipParameterMixin	Implements the interface for populating and filtering a compound relationship parameter model.
CompoundParameterDefinitionMixin	Handles signals from db mngr for parameter_definition models.
CompoundParameterValueMixin	Handles signals from db mngr for parameter_value models.
CompoundObjectParameterDefinitionModel	A model that concatenates several single object parameter_definition models
CompoundRelationshipParameterDefinitionModel	A model that concatenates several single relationship parameter_definition models
CompoundObjectParameterValueModel	A model that concatenates several single object parameter_value models
CompoundRelationshipParameterValueModel	A model that concatenates several single relationship parameter_value models

class spinetoolbox.spine_db_editor.mvcmodels.compound_parameter_models.**CompoundParameterModel**(*parent*, *db_mngr*, **db_maps*)

Bases: *spinetoolbox.helpers.FetchParent*, *spinetoolbox.mvcmodels.compound_table_model.CompoundWithEmptyTableModel*

A model that concatenates several single parameter models and one empty parameter model.

Initializes model.

Parameters

- **parent** (*SpineDBEditor*) – the parent object
- **db_mngr** (*SpineDBManager*) – the database manager
- ***db_maps** (*DiffDatabaseMapping*) – the database maps included in the model

property fetch_item_type

Returns the type of item to fetch, e.g., "object_class". Used to create an initial query for this item.

Returns
str

abstract property entity_class_type

Returns the entity_class type, either 'object_class' or 'relationship_class'.

Returns
str

abstract property item_type

Returns the parameter item type, either 'parameter_definition' or 'parameter_value'.

Returns
str

property _single_model_type

Returns a constructor for the single models.

Returns
SingleParameterModel

property _empty_model_type

Returns a constructor for the empty model.

Returns
EmptyParameterModel

property entity_class_id_key

Returns the key corresponding to the entity_class id (either "object_class_id" or "relationship_class_id")

Returns
str

property parameter_definition_id_key

canFetchMore(*_parent*)

Returns True if any of the submodels that haven't been fetched yet can fetch more.

fetchMore(*_parent*)

Fetches the next sub model and increments the fetched counter.

filter_query(*query*, *subquery*, *db_map*)

Filters the initial query created using the `fetch_item_type` property.

Parameters

- **query** (*Query*) – The query
- **subquery** (*Alias*) – The source of the query
- **db_map** (*DiffDatabaseMapping*) –

Returns
Query

abstract _make_header()

init_model()

Initializes the model.

_make_auto_filter_menus()

Makes auto filter menus.

get_auto_filter_menu(*logical_index*)

Returns auto filter menu for given logical index from header view.

Parameters
logical_index (*int*) –

Returns
ParameterViewFilterMenu

_modify_data_in_filter_menus(*action*, *db_map*, *db_items*)

Modifies data in filter menus.

Parameters

- **action** (*str*) – either 'add', 'remove', or 'update'

- **db_map** (*DiffDatabaseMapping*) –
- **db_items** (*list(dict)*) –

_do_add_data_to_filter_menus(*db_map*, *db_items*)

_do_update_data_in_filter_menus(*db_map*, *db_items*)

_do_remove_data_from_filter_menus(*db_map*, *db_items*)

headerData(*section*, *orientation=Qt.Horizontal*, *role=Qt.DisplayRole*)

Returns an italic font in case the given column has an autofilter installed.

_create_empty_model()

Returns the empty model for this compound model.

Returns
EmptyParameterModel

filter_accepts_model(*model*)

Returns a boolean indicating whether or not the given model passes the filter for compound model.

Parameters
model (SingleParameterModel, EmptyParameterModel) –

Returns
bool

_class_filter_accepts_model(*model*)

_auto_filter_accepts_model(*model*)

accepted_single_models()

Returns a list of accepted single models by calling filter_accepts_model on each of them, just for convenience.

Returns
list

_invalidate_filter()

Sets the filter invalid.

stop_invalidating_filter()

Stops invalidating the filter.

set_filter_class_ids(*class_ids*)

clear_auto_filter()

set_auto_filter(*field*, *values*)

Updates and applies the auto filter.

Parameters

- **field** (*str*) – the field name
- **values** (*dict*) – mapping db_map to entity_class id to accepted values for the field

_set_compound_auto_filter(*field*, *values*)

Sets the auto filter for given column in the compound model.

Parameters

- **field** (*str*) – the field name
- **values** (*dict*) – maps tuple (database map, entity_class id) to list of accepted ids for the field

_set_single_auto_filter(*model*, *field*)

Sets the auto filter for given column in the given single model.

Parameters

- **model** (SingleParameterModel) – the model
- **field** (*str*) – the field name

Returns
True if the auto-filtered values were updated, None otherwise

Return type
bool

_row_map_iterator_for_model(*model*)

Yields row map for the given model. Reimplemented to take filter status into account.

Parameters
model (SingleParameterModel, EmptyParameterModel) –

Returns
tuples (model, row number) for each accepted row

Return type
list

_models_with_db_map(*db_map*)

Returns a collection of single models with given db_map.

Parameters
db_map (*DiffDatabaseMapping*) –

Returns
list

receive_entity_classes_removed(*db_map_data*)

Runs when entity classes are removed from the dbs. Removes sub-models for the given entity classes and dbs.

Parameters
db_map_data (*dict*) – list of removed dict-items keyed by DiffDatabaseMapping

_items_per_class(*items*)

Returns a dict mapping entity_class ids to a set of items.

Parameters
items (*list*) –

Returns
dict

receive_parameter_data_added(*db_map_data*)

Runs when either parameter definitions or values are added to the dbs. Adds necessary sub-models and initializes them with data. Also notifies the empty model so it can remove rows that are already in.

Parameters
db_map_data (*dict*) – list of added dict-items keyed by DiffDatabaseMapping

_get_insert_position(*model*)

_create_single_model(*db_map*, *entity_class_id*, *committed*)

_add_parameter_data(*db_map*, *entity_class_id*, *ids*, *committed*)

Creates new single model and resets it with the given parameter ids.

Parameters

- **db_map** (*DiffDatabaseMapping*) – database map
- **entity_class_id** (*int*) – parameter's entity class id
- **ids** (*list of int*) – parameter ids
- **committed** (*bool*) – True if the ids have been committed, False otherwise

receive_parameter_data_updated(*db_map_data*)

Runs when either parameter definitions or values are updated in the dbs. Emits dataChanged so the parameter_name column is refreshed.

Parameters
db_map_data (*dict*) – list of updated dict-items keyed by DiffDatabaseMapping

receive_parameter_data_removed(*db_map_data*)

Runs when either parameter definitions or values are removed from the dbs. Removes the affected rows from the corresponding single models.

Parameters
db_map_data (*dict*) – list of removed dict-items keyed by DiffDatabaseMapping

_emit_data_changed_for_column(*field*)

Lazily emits data changed for an entire column.

Parameters
field (*str*) – the column header

db_item(*index*)

db_map_id(*index*)

index_name(*index*)

Generates a name for data at given index.

Parameters
index (*QModelIndex*) – index to model

Returns
label identifying the data

Return type
str

get_set_data_delayed(*index*)

Returns a function that ParameterValueEditor can call to set data for the given index at any later time, even if the model changes.

Parameters
index (*QModelIndex*) –

Returns
function

get_entity_class_id(*index*, *db_map*)

filter_by(*rows_per_column*)

filter_excluding(*rows_per_column*)

class spinetoolbox.spine_db_editor.mvcmodels.compound_parameter_models.**CompoundObjectParameterMixin**

Implements the interface for populating and filtering a compound object parameter model.

property entity_class_type

class spinetoolbox.spine_db_editor.mvcmodels.compound_parameter_models.**CompoundRelationshipParameterMixin**

Implements the interface for populating and filtering a compound relationship parameter model.

property entity_class_type

class spinetoolbox.spine_db_editor.mvcmodels.compound_parameter_models.**CompoundParameterDefinitionMixin**

Handles signals from db mngr for parameter_definition models.

property item_type

class spinetoolbox.spine_db_editor.mvcmodels.compound_parameter_models.**CompoundParameterValueMixin**(**args*, ***kwa*

Handles signals from db mngr for parameter_value models.

property item_type

abstract property entity_type

Returns the entity type, either 'object' or 'relationship' Used by update_single_main_filter.

Returns
str

init_model()

set_filter_entity_ids(*entity_ids*)

set_filter_alternative_ids(*alternative_ids*)

_create_single_model(*db_map*, *entity_class_id*, *committed*)

receive_alternatives_updated(*db_map_data*)

Updated alternative column

Parameters
db_map_data (*dict*) – list of updated dict-items keyed by DiffDatabaseMapping

class spinetoolbox.spine_db_editor.mvcmodels.compound_parameter_models.**CompoundObjectParameterDefinitio**

Bases: *CompoundObjectParameterMixin*, *CompoundParameterDefinitionMixin*, *CompoundParameterModel*

A model that concatenates several single object parameter_definition models and one empty object parameter_definition model.

Initializes model.

Parameters

- **parent** (SpineDBEditor) – the parent object
- **db_mngr** (SpineDBManager) – the database manager
- ***db_maps** (*DiffDatabaseMapping*) – the database maps included in the model

_make_header()

class spinetoolbox.spine_db_editor.mvcmodels.compound_parameter_models.**CompoundRelationshipParameterDef**

Bases: *CompoundRelationshipParameterMixin*, *CompoundParameterDefinitionMixin*, *CompoundParameterModel*

A model that concatenates several single relationship parameter_definition models and one empty relationship parameter_definition model.

Initializes model.

Parameters

- **parent** (SpineDBEditor) – the parent object
- **db_mngr** (SpineDBManager) – the database manager
- ***db_maps** (*DiffDatabaseMapping*) – the database maps included in the model

_make_header()

class spinetoolbox.spine_db_editor.mvcmodels.compound_parameter_models.**CompoundObjectParameterValueMode**

Bases: *CompoundObjectParameterMixin*, *CompoundParameterValueMixin*, *CompoundParameterModel*

A model that concatenates several single object parameter_value models and one empty object parameter_value model.

property entity_type

Returns the entity type, either 'object' or 'relationship' Used by update_single_main_filter.

Returns
str

_make_header()

class spinetoolbox.spine_db_editor.mvcmodels.compound_parameter_models.**CompoundRelationshipParameterVal**

Bases: *CompoundRelationshipParameterMixin*, *CompoundParameterValueMixin*, *CompoundParameterModel*

A model that concatenates several single relationship parameter_value models and one empty relationship parameter_value model.

property entity_type

Returns the entity type, either 'object' or 'relationship' Used by update_single_main_filter.

Returns
str

_make_header()

spinetoolbox.spine_db_editor.mvcmodels.empty_parameter_models

Empty models for parameter definitions and values.

authors

M. Marin (KTH)

date

28.6.2019

Module Contents

Classes

EmptyParameterModel	An empty parameter model.
EmptyParameterDefinitionModel	An empty parameter_definition model.
EmptyObjectParameterDefinitionModel	An empty object parameter_definition model.
EmptyRelationshipParameterDefinitionModel	An empty relationship parameter_definition model.
EmptyParameterValueModel	An empty parameter_value model.
EmptyObjectParameterValueModel	An empty object parameter_value model.
EmptyRelationshipParameterValueModel	An empty relationship parameter_value model.

class spinetoolbox.spine_db_editor.mvcmodels.empty_parameter_models.**EmptyParameterModel**(*parent*, *header*, *db_mngr*)

Bases: *spinetoolbox.mvcmodels.empty_row_model.EmptyRowModel*

An empty parameter model.

Initialize class.

Parameters

- **parent** (*Object*) – the parent object, typically a CompoundParameterModel
- **header** (*list*) – list of field names for the header
- **db_mngr** (SpineDBManager) –

abstract property item_type

The item type, either 'parameter_value' or 'parameter_definition', required by the value_field property.

abstract property entity_class_type

Either 'object_class' or 'relationship_class'.

property entity_class_id_key

property entity_class_name_key

property can_be_filtered

property value_field

accepted_rows()

db_item(*_index*)

item_id(*_row*)

data(*index*, *role=Qt.DisplayRole*)

Returns the data stored under the given role for the item referred to by the index.

Parameters

- **index** (*QModelIndex*) – Index of item
- **role** (*int*) – Data role

Returns
Item data for given role.

_make_unique_id(*item*)

Returns a unique id for the given model item (name-based). Used by receive_parameter_data_added.

receive_parameter_data_added(*db_map_data*)

Runs when parameter definitions or values are added. Finds and removes model items that were successfully added to the db.

batch_set_data(*indexes*, *data*)

Sets data for indexes in batch. If successful, add items to db.

abstract add_items_to_db(*db_map_data*)

Add items to db.

Parameters
db_map_data (*dict*) – mapping DiffDatabaseMapping instance to list of items

_make_db_map_data(*rows*)

Returns model data grouped by database map.

Parameters
rows (*set*) – group data from these rows

Returns
mapping DiffDatabaseMapping instance to list of items

Return type
dict

class spinetoolbox.spine_db_editor.mvcmodels.empty_parameter_models.**EmptyParameterDefinitionModel**(**args*, ***kwa*

Bases: *spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.FillInValueListIdMixin*, *spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.FillInEntityClassIdMixin*, *spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.FillInParameterNameMixin*, *EmptyParameterModel*

An empty parameter_definition model.

Initializes lookup dicts.

property item_type

The item type, either 'parameter_value' or 'parameter_definition', required by the value_field property.

abstract property entity_class_type

See base class.

add_items_to_db(*db_map_data*)

See base class.

_check_item(*item*)

Checks if a db item is ready to be inserted.

class spinetoolbox.spine_db_editor.mvcmodels.empty_parameter_models.**EmptyObjectParameterDefinitionModel**

Bases: *EmptyParameterDefinitionModel*

An empty object parameter_definition model.

Initializes lookup dicts.

property entity_class_type

See base class.

class spinetoolbox.spine_db_editor.mvcmodels.empty_parameter_models.**EmptyRelationshipParameterDefinitio**

Bases: *EmptyParameterDefinitionModel*

An empty relationship parameter_definition model.

Initializes lookup dicts.

property entity_class_type

See base class.

flags(*index*)

Additional hack to make the object_class_name_list column non-editable.

class spinetoolbox.spine_db_editor.mvcmodels.empty_parameter_models.**EmptyParameterValueModel**(**args*, ***kwargs*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.InferEntityClassIdMixin*, *spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.FillInAlternativeIdMixin*, *spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.FillInParameterDefinitionIdsMixin*, *spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.FillInEntityIdsMixin*, *spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.FillInEntityClassIdMixin*, *EmptyParameterModel*

An empty parameter_value model.

Initializes lookup dicts.

property item_type

The item type, either 'parameter_value' or 'parameter_definition', required by the value_field property.

abstract property entity_type

Either 'object' or "relationship'.

property entity_id_key

property entity_name_key

property entity_name_key_in_cache

_make_unique_id(*item*)

Returns a unique id for the given model item (name-based). Used by receive_parameter_data_added.

add_items_to_db(*db_map_data*)

See base class.

_check_item(*db_map*, *item*)

Checks if a db item is ready to be inserted.

class spinetoolbox.spine_db_editor.mvcmodels.empty_parameter_models.**EmptyObjectParameterValueModel**(**args*, ***kw*

Bases: *EmptyParameterValueModel*

An empty object parameter_value model.

Initializes lookup dicts.

property entity_class_type

Either 'object_class' or 'relationship_class'.

property entity_type

Either 'object' or "relationship'.

class spinetoolbox.spine_db_editor.mvcmodels.empty_parameter_models.**EmptyRelationshipParameterValueModel**

Bases: *spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.MakeRelationshipOnTheFlyMixin*, *EmptyParameterValueModel*

An empty relationship parameter_value model.

Initializes lookup dicts.

property entity_class_type

Either 'object_class' or 'relationship_class'.

property entity_type

Either 'object' or "relationship'.

_add_entities_on_the_fly = True

add_items_to_db(*db_map_data*)

See base class.

spinetoolbox.spine_db_editor.mvcmodels.entity_tree_item

Classes to represent entities in a tree.

authors

P. Vennström (VTT), M. Marin (KTH)

date

11.3.2019

Module Contents

Classes

EntityRootItem	A tree item that may belong in multiple databases.
ObjectTreeRootItem	An object tree root item.
RelationshipTreeRootItem	A relationship tree root item.
EntityClassItem	An entity_class item.
ObjectClassItem	An object_class item.
RelationshipClassItem	A relationship_class item.
ObjectRelationshipClassItem	A relationship_class item.
MemberObjectClassItem	A member object class item.
EntityItem	An entity item.
ObjectItem	An object item.
MemberObjectItem	A member object item.
RelationshipItem	A relationship item.

class spinetoolbox.spine_db_editor.mvcmodels.entity_tree_item.**EntityRootItem**(*model=None*, *db_map_ids=None*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.multi_db_tree_item.MultiDBTreeItem*

A tree item that may belong in multiple databases.

Init class.

Parameters

- **model** (*MinimalTreeModel, optional*) – item's model
- **db_map_ids** (*dict, optional*) – maps instances of DiffDatabaseMapping to the id of the item in that db

property display_id

See super class.

property display_icon

Returns an icon to display next to the name. Reimplement in subclasses to return something nice.

property display_data

See super class.

item_type = root

set_data(*column*, *value*, *role*)

See base class.

class spinetoolbox.spine_db_editor.mvcmodels.entity_tree_item.**ObjectTreeRootItem**(*model=None*, *db_map_ids=None*)

Bases: *EntityRootItem*

An object tree root item.

Init class.

Parameters

- **model** (*MinimalTreeModel, optional*) – item's model
- **db_map_ids** (*dict, optional*) – maps instances of DiffDatabaseMapping to the id of the item in that db

property child_item_class

Returns ObjectClassItem.

item_type = root

class spinetoolbox.spine_db_editor.mvcmodels.entity_tree_item.**RelationshipTreeRootItem**(*model=None*, *db_map_ids=None*)

Bases: *EntityRootItem*

A relationship tree root item.

Init class.

Parameters

- **model** (*MinimalTreeModel, optional*) – item's model
- **db_map_ids** (*dict, optional*) – maps instances of DiffDatabaseMapping to the id of the item in that db

property child_item_class

Returns RelationshipClassItem.

item_type = root

class spinetoolbox.spine_db_editor.mvcmodels.entity_tree_item.**EntityClassItem**(**args*, ***kwargs*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.multi_db_tree_item.MultiDBTreeItem*

An entity_class item.

Overridden method to declare group_child_count attribute.

property display_icon

Returns class icon.

_display_icon(*for_group=False*)

data(*column*, *role=Qt.DisplayRole*)

Returns data for given column and role.

remove_children(*position*, *count*)

Overriden method to keep the group child count up to date.

filter_query(*query*, *subquery*, *db_map*)

Filters the initial query created using the `fetch_item_type` property.

Parameters

- **query** (*Query*) – The query
- **subquery** (*Alias*) – The source of the query
- **db_map** (*DiffDatabaseMapping*) –

Returns

Query

set_data(*column*, *value*, *role*)

See base class.

class spinetoolbox.spine_db_editor.mvcmodels.entity_tree_item.**ObjectClassItem**(**args*, ***kwargs*)

Bases: *EntityClassItem*

An object_class item.

Overridden method to declare group_child_count attribute.

property child_item_class

Returns ObjectItem.

property _children_sort_key

Reimplemented so groups are above non-groups.

item_type = object_class

default_parameter_data()

Return data to put as default in a parameter table when this item is selected.

class spinetoolbox.spine_db_editor.mvcmodels.entity_tree_item.**RelationshipClassItem**(**args*, ***kwargs*)

Bases: *EntityClassItem*

A relationship_class item.

Overridden method to declare group_child_count attribute.

property child_item_class

Returns RelationshipItem.

visual_key = ['name', 'object_class_name_list']

item_type = relationship_class

default_parameter_data()

Return data to put as default in a parameter table when this item is selected.

class spinetoolbox.spine_db_editor.mvcmodels.entity_tree_item.**ObjectRelationshipClassItem**(**args*, ***kwargs*)

Bases: *RelationshipClassItem*

A relationship_class item.

Overridden method to declare group_child_count attribute.

set_data(*column*, *value*, *role*)

See base class.

filter_query(*query*, *subquery*, *db_map*)

Filters the initial query created using the `fetch_item_type` property.

Parameters

- **query** (*Query*) – The query
- **subquery** (*Alias*) – The source of the query
- **db_map** (*DiffDatabaseMapping*) –

Returns

Query

class spinetoolbox.spine_db_editor.mvcmodels.entity_tree_item.**MemberObjectClassItem**(**args*, ***kwargs*)

Bases: *ObjectClassItem*

A member object class item.

Overridden method to declare group_child_count attribute.

property display_id

Returns an id for display based on the display key. This id must be the same across all db_maps. If it's not, this property becomes None and measures need to be taken (see update_children_by_id).

property display_data

Returns the name for display.

property child_item_class

Returns MemberObjectItem.

item_type = members

db_map_data(*db_map*)

Returns data for this item as if it was indeed an object class.

_display_icon(*for_group=False*)

Returns icon for this item as if it was indeed an object class.

filter_query(*query*, *subquery*, *db_map*)

Filters the initial query created using the `fetch_item_type` property.

Parameters

- **query** (*Query*) – The query
- **subquery** (*Alias*) – The source of the query
- **db_map** (*DiffDatabaseMapping*) –

Returns
Query

default_parameter_data()

Return data to put as default in a parameter table when this item is selected.

data(*column*, *role=Qt.DisplayRole*)

Returns data for given column and role.

class spinetoolbox.spine_db_editor.mvcmodels.entity_tree_item.**EntityItem**(**args*, ***kwargs*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.multi_db_tree_item.MultiDBTreeItem*

An entity item.

Init class.

Parameters

- **model** (*MinimalTreeModel, optional*) – item's model
- **db_map_ids** (*dict, optional*) – maps instances of DiffDatabaseMapping to the id of the item in that db

property members_item

property display_icon

Returns corresponding class icon.

is_group()

data(*column*, *role=Qt.DisplayRole*)

Returns data for given column and role.

set_data(*column*, *value*, *role*)

See base class.

_can_fetch_members_item()

_fetch_members_item()

can_fetch_more()

Returns whether or not this item can fetch more.

fetch_more()

Fetches children from all associated databases.

class spinetoolbox.spine_db_editor.mvcmodels.entity_tree_item.**ObjectItem**(**args*, ***kwargs*)

Bases: *EntityItem*

An object item.

Init class.

Parameters

- **model** (*MinimalTreeModel, optional*) – item's model
- **db_map_ids** (*dict, optional*) – maps instances of DiffDatabaseMapping to the id of the item in that db

property child_item_class

Child class is always *ObjectRelationshipClassItem*.

item_type = object

default_parameter_data()

Return data to put as default in a parameter table when this item is selected.

filter_query(*query*, *subquery*, *db_map*)

Filters the initial query created using the `fetch_item_type` property.

Parameters

- **query** (*Query*) – The query
- **subquery** (*Alias*) – The source of the query
- **db_map** (*DiffDatabaseMapping*) –

Returns

Query

class spinetoolbox.spine_db_editor.mvcmodels.entity_tree_item.**MemberObjectItem**(**args*, ***kwargs*)

Bases: *ObjectItem*

A member object item.

Init class.

Parameters

- **model** (*MinimalTreeModel, optional*) – item's model
- **db_map_ids** (*dict, optional*) – maps instances of DiffDatabaseMapping to the id of the item in that db

property display_icon

Returns corresponding class icon.

property display_data

"Returns the name for display.

item_type = entity_group

visual_key = ['member_name']

has_children()

Returns whether or not this item has or could have children.

can_fetch_more()

Returns whether or not this item can fetch more.

class spinetoolbox.spine_db_editor.mvcmodels.entity_tree_item.**RelationshipItem**(**args, **kwargs*)

Bases: *EntityItem*

A relationship item.

Overridden method to make sure we never try to fetch this item.

property object_name_list

property display_data

"Returns the name for display.

property edit_data

visual_key = ['name', 'object_name_list']

item_type = relationship

default_parameter_data()

Return data to put as default in a parameter table when this item is selected.

has_children()

Returns whether or not this item has or could have children.

can_fetch_more()

Returns whether or not this item can fetch more.

is_valid()

Checks that the grand parent object is still in the relationship.

spinetoolbox.spine_db_editor.mvcmodels.entity_tree_models

Models to represent entities in a tree.

authors

P. Vennström (VTT), M. Marin (KTH)

date

11.3.2019

Module Contents

Classes

ObjectTreeModel	An 'object-oriented' tree model.
RelationshipTreeModel	A relationship-oriented tree model.

class spinetoolbox.spine_db_editor.mvcmodels.entity_tree_models.**ObjectTreeModel**(*parent*, *db_mngr*, **db_maps*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.multi_db_tree_model.MultiDBTreeModel*

An 'object-oriented' tree model.

Init class.

Parameters

- **parent** (SpineDBEditor) –
- **db_mngr** (SpineDBManager) – A manager for the given db_maps
- **db_maps** (*iter*) – DiffDatabaseMapping instances

property root_item_type

Implement in subclasses to create a model specific to any entity type.

_parent_object_data(*db_map_data*)

Takes given object data and returns the same data keyed by parent tree-item.

Parameters

db_map_data (*dict*) – maps DiffDatabaseMapping instances to list of items as dict

Returns

maps parent tree-items to DiffDatabaseMapping instances to list of item ids

Return type

dict

_parent_relationship_class_data(*db_map_data*)

Takes given relationship_class data and returns the same data keyed by parent tree-item.

Parameters

db_map_data (*dict*) – maps DiffDatabaseMapping instances to list of items as dict

Returns

maps parent tree-items to DiffDatabaseMapping instances to list of item ids

Return type
dict

`_parent_relationship_data`(*db_map_data*)

Takes given relationship data and returns the same data keyed by parent tree-item.

Parameters
db_map_data (*dict*) – maps DiffDatabaseMapping instances to list of items as dict

Returns
maps parent tree-items to DiffDatabaseMapping instances to list of item ids

Return type
dict

`_parent_relationship_data_for_update`(*db_map_data*)

Takes given relationship data and returns the same data keyed by parent tree-item.

Parameters
db_map_data (*dict*) – maps DiffDatabaseMapping instances to list of items as dict

Returns
maps parent tree-items to DiffDatabaseMapping instances to list of item ids

Return type
dict

`_parent_entity_member_data`(*db_map_data*)

Takes given entity member data and returns the same data keyed by parent tree-item.

Parameters
db_map_data (*dict*) – maps DiffDatabaseMapping instances to list of items as dict

Returns
maps parent tree-items to DiffDatabaseMapping instances to list of item ids

Return type
dict

`add_object_classes`(*db_map_data*)

`add_objects`(*db_map_data*)

`add_relationship_classes`(*db_map_data*)

`add_relationships`(*db_map_data*)

`add_entity_groups`(*db_map_data*)

`remove_object_classes`(*db_map_data*)

`remove_objects`(*db_map_data*)

`remove_relationship_classes`(*db_map_data*)

`remove_relationships`(*db_map_data*)

`remove_entity_groups`(*db_map_data*)

`update_object_classes`(*db_map_data*)

`update_objects`(*db_map_data*)

update_relationship_classes(*db_map_data*)

update_relationships(*db_map_data*)

find_next_relationship_index(*index*)

Find and return next occurrence of relationship item.

class spinetoolbox.spine_db_editor.mvcmodels.entity_tree_models.**RelationshipTreeModel**(*parent*, *db_mngr*, **db_maps*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.multi_db_tree_model.MultiDBTreeModel*

A relationship-oriented tree model.

Init class.

Parameters

- **parent** (SpineDBEditor) –
- **db_mngr** (SpineDBManager) – A manager for the given db_maps
- **db_maps** (*iter*) – DiffDatabaseMapping instances

property root_item_type

Implement in subclasses to create a model specific to any entity type.

_parent_relationship_data(*db_map_data*)

Takes given relationship data and returns the same data keyed by parent tree-item.

Parameters
db_map_data (*dict*) – maps DiffDatabaseMapping instances to list of items as dict

Returns
maps parent tree-items to DiffDatabaseMapping instances to list of item ids

Return type
dict

add_relationship_classes(*db_map_data*)

add_relationships(*db_map_data*)

remove_relationship_classes(*db_map_data*)

remove_relationships(*db_map_data*)

update_relationship_classes(*db_map_data*)

update_relationships(*db_map_data*)

spinetoolbox.spine_db_editor.mvcmodels.frozen_table_model

Contains FrozenTableModel class.

author

P. Vennström (VTT)

date
24.9.2019

Module Contents

Classes

FrozenTableModel	Used by custom_qtableview.FrozenTableView

class spinetoolbox.spine_db_editor.mvcmodels.frozen_table_model.**FrozenTableModel**(*parent*, *headers=None*, *data=None*)

Bases: PySide2.QtCore.QAbstractItemModel

Used by custom_qtableview.FrozenTableView

Parameters
parent (TabularViewMixin) –

property headers

parent(*child=None*)

index(*row*, *column*, *parent=QModelIndex()*)

reset_model(*data*, *headers*)

clear_model()

rowCount(*parent=QModelIndex()*)

columnCount(*parent=QModelIndex()*)

row(*index*)

data(*index*, *role*)

headerData(*section*, *orientation*, *role=Qt.DisplayRole*)

spinetoolbox.spine_db_editor.mvcmodels.item_metadata_table_model

Contains *ItemMetadataTableModel* and associated functionality.

author

A. Soininen (VTT)

date
25.3.2022

Module Contents

Classes

ExtraColumn	Identifiers for hidden table columns.
ItemType	Allowed item types.
ItemMetadataTableModel	Model for entity and parameter value metadata.

class spinetoolbox.spine_db_editor.mvcmodels.item_metadata_table_model.**ExtraColumn**

Bases: enum.IntEnum

Identifiers for hidden table columns.

Initialize self. See help(type(self)) for accurate signature.

ITEM_METADATA_ID

METADATA_ID

class spinetoolbox.spine_db_editor.mvcmodels.item_metadata_table_model.**ItemType**

Bases: enum.Enum

Allowed item types.

ENTITY

VALUE

class spinetoolbox.spine_db_editor.mvcmodels.item_metadata_table_model.**ItemMetadataTableModel**(*db_mngr*, *db_maps*, *parent=None*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.metadata_table_model_base.MetadataTableModelBase*

Model for entity and parameter value metadata.

Parameters

- **db_mngr** (*SpineDBManager*) – database manager
- **db_maps** (*Iterable of DatabaseMappingBase*) – database maps
- **parent** (*QObject*) – parent object

_ITEM_NAME_KEY = metadata_name

_ITEM_VALUE_KEY = metadata_value

clear()

Clears the model.

static _make_hidden_adder_columns()

See base class.

set_entity_ids(*db_map_ids*)

Sets the model to show metadata from given entity.

Parameters

db_map_ids (*dict*) – mapping from database mapping to entity's id in that database

set_parameter_value_ids(*db_map_ids*)

Sets the model to show metadata from given parameter value.

Parameters

db_map_ids (*dict*) – mapping from database mapping to value's id in that database

_reset_metadata(*item_type*, *db_map_ids*, *metadata*)

Resets model.

Parameters

- **item_type** (ItemType) – current item type
- **db_map_ids** (*dict*) – mapping from database mapping to value's id in that database
- **metadata** (*dict*) – mapping from database mapping to metadata records

_add_data_to_db_mngr(*name*, *value*, *db_map*)

See base class.

_update_data_in_db_mngr(*id_*, *name*, *value*, *db_map*)

See base class

roll_back(*db_maps*)

Rolls back changes in database.

Parameters

db_maps (*Iterable of DiffDatabaseMapping*) – database mappings that have been rolled back

flags(*index*)

static _ids_from_added_item(*item*)

See base class.

static _extra_cells_from_added_item(*item*)

See base class.

_set_extra_columns(*row*, *ids*)

See base class.

_database_table_name()

See base class

_row_id(*row*)

See base class.

add_item_metadata(*db_map_data*)

Adds new item metadata from database manager to the model.

Parameters

db_map_data (*dict*) – added items keyed by database mapping

update_item_metadata(*db_map_data*)

Updates item metadata in model after it has been updated in databases.

Parameters

db_map_data (*dict*) – updated metadata records

remove_item_metadata(*db_map_data*)

Removes item metadata from model after it has been removed from databases.

Parameters
db_map_data (*dict*) – removed items keyed by database mapping

update_metadata(*db_map_data*)

Updates metadata.

Parameters
db_map_data (*dict*) – updated items keyed by database mapping

remove_metadata(*db_map_data*)

Removes entries that correspond to removed metadata.

Parameters
db_map_data (*dict*) – removed items keyed by database mapping

spinetoolbox.spine_db_editor.mvcmodels.metadata_table_model

Contains *MetadataTableModel* and associated functionality.

author

A. Soininen (VTT)

date
7.2.2022

Module Contents

Classes

ExtraColumn	Identifiers for hidden table columns.
MetadataTableModel	Model for metadata.

class spinetoolbox.spine_db_editor.mvcmodels.metadata_table_model.**ExtraColumn**

Bases: enum.IntEnum

Identifiers for hidden table columns.

Initialize self. See help(type(self)) for accurate signature.

ID

class spinetoolbox.spine_db_editor.mvcmodels.metadata_table_model.**MetadataTableModel**(*db_mngr*, *db_maps*, *parent=None*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.metadata_table_model_base.MetadataTableModelBase*, *spinetoolbox.helpers.FetchParent*

Model for metadata.

Parameters

- **db_mngr** (*SpineDBManager*) – database manager

- **db_maps** (*Iterable of DatabaseMappingBase*) – database maps
- **parent** (*QObject*) – parent object

property fetch_item_type

Returns the type of item to fetch, e.g., "object_class". Used to create an initial query for this item.

Returns
str

_ITEM_NAME_KEY = name

_ITEM_VALUE_KEY = value

static _make_hidden_adder_columns()

See base class.

_add_data_to_db_mngr(*name*, *value*, *db_map*)

See base class.

_update_data_in_db_mngr(*id_*, *name*, *value*, *db_map*)

See base class

roll_back(*db_maps*)

Rolls back changes in database.

Parameters
db_maps (*Iterable of DiffDatabaseMapping*) – database mappings that have been rolled back

_database_table_name()

See base class

_row_id(*row*)

See base class.

flags(*index*)

canFetchMore(_)

fetchMore(_)

static _ids_from_added_item(*item*)

See base class.

static _extra_cells_from_added_item(*item*)

See base class.

_set_extra_columns(*row*, *ids*)

See base class.

add_metadata(*db_map_data*)

Adds new metadata from database manager to the model.

Parameters
db_map_data (*dict*) – added metadata items keyed by database mapping

update_metadata(*db_map_data*)

Updates model according to data received from database manager.

Parameters
db_map_data (*dict*) – updated metadata items keyed by database mapping

remove_metadata(*db_map_data*)

Removes metadata from model after it has been removed from databases.

Parameters
db_map_data (*dict*) – removed items keyed by database mapping

add_and_update_metadata(*db_map_data*)

Adds and updates metadata after changes in database.

Parameters
db_map_data (*dict*) – changed items keyed by database mapping

spinetoolbox.spine_db_editor.mvcmodels.metadata_table_model_base

Contains base class for metadata table models associated functionality.

author

A. Soininen (VTT)

date
25.4.2022

Module Contents

Classes

Column	Identifiers for visible table columns.
MetadataTableModelBase	Base for metadata table models

Attributes

FLAGS_FIXED
FLAGS_EDITABLE

class spinetoolbox.spine_db_editor.mvcmodels.metadata_table_model_base.**Column**

Bases: enum.IntEnum

Identifiers for visible table columns.

Initialize self. See help(type(self)) for accurate signature.

NAME = 0

VALUE = 1

DB_MAP = 2

static max()

spinetoolbox.spine_db_editor.mvcmodels.metadata_table_model_base.**FLAGS_FIXED**

spinetoolbox.spine_db_editor.mvcmodels.metadata_table_model_base.**FLAGS_EDITABLE**

class spinetoolbox.spine_db_editor.mvcmodels.metadata_table_model_base.**MetadataTableModelBase**(*db_mngr*, *db_maps*, *parent=None*)

Bases: PySide2.QtCore.QAbstractTableModel

Base for metadata table models

Parameters

- **db_mngr** (SpineDBManager) – database manager
- **db_maps** (*Iterable of DatabaseMappingBase*) – database maps
- **parent** (*QObject*) – parent object

msg_error

Emitted when an error occurs.

_HEADER = ['name', 'value', 'database']

_ITEM_NAME_KEY

_ITEM_VALUE_KEY

classmethod _make_adder_row(*default_db_map*)

Generates a new empty last row.

Parameters
default_db_map (*DiffDatabaseMapping*) – initial database mapping

Returns
empty row

Return type
list

abstract static _make_hidden_adder_columns()

Creates hidden extra columns for adder row.

Returns
extra columns

Return type
list

set_db_maps(*db_maps*)

Changes current database mappings.

Parameters
db_maps (*Iterable of DiffDatabaseMapping*) – database mappings

rowCount(*parent=QModelIndex()*)

columnCount(*parent=QModelIndex()*)

data(*index*, *role=Qt.DisplayRole*)

abstract _add_data_to_db_mngr(*name*, *value*, *db_map*)

Tells database manager to start adding data.

Parameters

- **name** (*str*) – metadata name
- **value** (*str*) – metadata value
- **db_map** (*DiffDatabaseMapping*) – database mapping

abstract _update_data_in_db_mngr(*id_*, *name*, *value*, *db_map*)

Tells database manager to start updating data.

Parameters

- **id** (*int*) – database id
- **name** (*str*) – metadata name
- **value** (*str*) – metadata value
- **db_map** (*DiffDatabaseMapping*) – database mapping

setData(*index*, *value*, *role=Qt.EditRole*)

batch_set_data(*indexes*, *values*)

Sets data in multiple indexes simultaneously.

Parameters

- **indexes** (*Iterable of QModelIndex*) – indexes to set
- **values** (*Iterable of str*) – values corresponding to indexes

headerData(*section*, *orientation*, *role=Qt.DisplayRole*)

insertRows(*row*, *count*, *parent=QModelIndex()*)

abstract _database_table_name()

Returns primary database table name.

Returns
table name

Return type
str

abstract _row_id(*row*)

Returns a unique row id.

Parameters
row (*list*) – data table row

Returns
id or None

Return type
int

removeRows(*row*, *count*, *parent=QModelIndex()*)

abstract static _ids_from_added_item(*item*)

Returns ids that uniquely identify an added database item.

Parameters
item (*dict*) – added item

Returns
unique identifier

Return type
Any

abstract static _extra_cells_from_added_item(*item*)

Constructs extra cells for data row from added database item.

Parameters
item (*dict*) – added item

Returns
extra cells

Return type
list

abstract _set_extra_columns(*row*, *ids*)

Sets extra columns for data row.

Parameters

- **row** (*list*) – data row
- **ids** (*Any*) –

_add_data(*db_map_data*)

Adds new data from database manager to the model.

Parameters
db_map_data (*dict*) – added items keyed by database mapping

_update_data(*db_map_data*, *id_column*)

Update data table after database update.

Parameters

- **db_map_data** (*dict*) – updated items keyed by database mapping
- **id_column** (*int*) – column that contains item ids

_remove_data(*db_map_data*, *id_column*)

Removes data from model after it has been removed from databases.

Parameters

- **db_map_data** (*dict*) – removed items keyed by database mapping
- **id_column** (*int*) – column that contains item ids

sort(*column*, *order=Qt.AscendingOrder*)

_find_db_map(*codename*)

Finds database mapping with given codename.

Parameters
codename (*str*) – database mapping's code name

Returns
database mapping or None if not found

Return type
DiffDatabaseMapping

_reserved_metadata()

Collects metadata names and values that are already in database.

Returns
mapping from database mapping to metadata name and value

Return type
dict

spinetoolbox.spine_db_editor.mvcmodels.multi_db_tree_item

Base classes to represent items from multiple databases in a tree.

authors

P. Vennström (VTT), M. Marin (KTH)

date
17.6.2020

Module Contents

Classes

MultiDBTreeItem	A tree item that may belong in multiple databases.

class spinetoolbox.spine_db_editor.mvcmodels.multi_db_tree_item.**MultiDBTreeItem**(*model=None*, *db_map_ids=None*)

Bases: *spinetoolbox.helpers.FetchParent*, *spinetoolbox.mvcmodels.minimal_tree_model.TreeItem*

A tree item that may belong in multiple databases.

Init class.

Parameters

- **model** (*MinimalTreeModel, optional*) – item's model
- **db_map_ids** (*dict, optional*) – maps instances of DiffDatabaseMapping to the id of the item in that db

property db_mngr

property child_item_class

Returns the type of child items. Reimplement in subclasses to return something more meaningful.

property display_id

Returns an id for display based on the display key. This id must be the same across all db_maps. If it's not, this property becomes None and measures need to be taken (see update_children_by_id).

property display_data

Returns the name for display.

property display_database

Returns the database for display.

property display_icon

Returns an icon to display next to the name. Reimplement in subclasses to return something nice.

property first_db_map

Returns the first associated db_map.

property last_db_map

Returns the last associated db_map.

property db_maps

Returns a list of all associated db_maps.

property db_map_ids

Returns dict with db_map as key and id as value

property _children_sort_key

property fetch_item_type

Returns the type of item to fetch, e.g., "object_class". Used to create an initial query for this item.

Returns
str

item_type

Item type identifier string. Should be set to a meaningful value by subclasses.

visual_key = ['name']

child_number()

Returns the rank of this item within its parent or -1 if it's an orphan.

abstract set_data(*column*, *value*, *role*)

Sets data for this item.

Parameters

- **column** (*int*) – column index
- **value** (*object*) – a new value
- **role** (*int*) – role of the new value

Returns
True if data was set successfully, False otherwise

Return type
bool

add_db_map_id(*db_map*, *id_*)

Adds id for this item in the given db_map.

take_db_map(*db_map*)

Removes the mapping for given db_map and returns it.

_deep_refresh_children()

Refreshes children after taking db_maps from them. Called after removing and updating children for this item.

deep_remove_db_map(*db_map*)

Removes given db_map from this item and all its descendants.

deep_take_db_map(*db_map*)

Removes given db_map from this item and all its descendants, and returns a new item from the db_map's data.

Returns
MultiDBTreeItem, NoneType

deep_merge(*other*)

Merges another item and all its descendants into this one.

db_map_id(*db_map*)

Returns the id for this item in given db_map or None if not present.

db_map_data(*db_map*)

Returns data for this item in given db_map or None if not present.

db_map_data_field(*db_map*, *field*, *default=None*)

Returns field from data for this item in given db_map or None if not found.

_create_new_children(*db_map*, *children_ids*)

Creates new items from ids associated to a db map.

Parameters

- **db_map** (*DiffDatabaseMapping*) – create children for this db_map
- **children_ids** (*iter*) – create children from these ids

Returns
new children

Return type
list of *MultiDBTreeItem*

_merge_children(*new_children*)

Merges new children into this item. Ensures that each child has a valid display id afterwards.

_insert_children_sorted(*new_children*)

Inserts and sorts children.

fetch_status_change()

Notifies the view that the model's layout has changed. This triggers a repaint so this item may be painted gray if no children.

can_fetch_more()

Returns whether or not this item can fetch more.

fetch_more()

Fetches children from all associated databases.

fetch_more_if_possible()

get_children_ids()

append_children_by_id(*db_map_ids*)

Appends children by id.

Parameters
db_map_ids (*dict*) – maps DiffDatabaseMapping instances to list of ids

remove_children_by_id(*db_map_ids*)

Removes children by id.

Parameters
db_map_ids (*dict*) – maps DiffDatabaseMapping instances to list of ids

is_valid()

Checks if the item is still valid after an update operation.

update_children_by_id(*db_map_ids*)

Updates children by id. Essentially makes sure all children have a valid display id after updating the underlying data. These may require 'splitting' a child into several for different dbs or merging two or more children from different dbs.

Examples of problems:

- The user renames an object_class in one db but not in the others –> we need to split
- The user renames an object_class and the new name is already 'taken' by another object_class in another db_map –> we need to merge

Parameters
db_map_ids (*dict*) – maps DiffDatabaseMapping instances to list of ids

insert_children(*position*, *children*)

Insert new children at given position. Returns a boolean depending on how it went.

Parameters

- **position** (*int*) – insert new items here
- **children** (*iter*) – insert items from this iterable

remove_children(*position*, *count*)

Removes count children starting from the given position.

clear_children()

Clear children list.

_refresh_child_map()

Recomputes the child map.

find_row(*db_map*, *id_*)

find_children_by_id(*db_map*, **ids*, *reverse=True*)

Generates children with the given ids in the given db_map. If the first id is None, then generates *all* children with the given db_map.

find_rows_by_id(*db_map*, **ids*, *reverse=True*)

_find_unsorted_rows_by_id(*db_map*, **ids*)

Generates rows corresponding to children with the given ids in the given db_map. If the only id given is None, then generates rows corresponding to *all* children with the given db_map.

data(*column*, *role=Qt.DisplayRole*)

Returns data for given column and role.

default_parameter_data()

Returns data to set as default in a parameter table when this item is selected.

spinetoolbox.spine_db_editor.mvcmodels.multi_db_tree_model

A base model class to represent items from multiple databases in a tree.

authors

P. Vennström (VTT), M. Marin (KTH)

date

17.6.2020

Module Contents

Classes

MultiDBTreeModel	Base class for all tree models in Spine db editor.

class spinetoolbox.spine_db_editor.mvcmodels.multi_db_tree_model.**MultiDBTreeModel**(*parent*, *db_mngr*, **db_maps*)

Bases: *spinetoolbox.mvcmodels.minimal_tree_model.MinimalTreeModel*

Base class for all tree models in Spine db editor.

Init class.

Parameters

- **parent** (SpineDBEditor) –
- **db_mngr** (SpineDBManager) – A manager for the given db_maps
- **db_maps** (*iter*) – DiffDatabaseMapping instances

abstract property root_item_type

Implement in subclasses to create a model specific to any entity type.

property root_item

property root_index

build_tree()

Builds tree.

columnCount(*parent=QModelIndex()*)

headerData(*section*, *orientation*, *role=Qt.DisplayRole*)

find_items(*db_map*, *path_prefix*, *fetch=False*)

Returns items at given path prefix.

spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins

Miscelaneous mixins for parameter models

authors

M. Marin (KTH)

date

4.10.2019

Module Contents

Classes

ConvertToDBMixin	Base class for all mixins that convert model items (name-based) into database items (id-based).
FillInAlternativeIdMixin	Fills in alternative names.
FillInParameterNameMixin	Fills in parameter names.
FillInValueListIdMixin	Fills in value list ids.
FillInEntityClassIdMixin	Fills in entity_class ids.
FillInEntityIdsMixin	Fills in entity ids.
FillInParameterDefinitionIdsMixin	Fills in parameter_definition ids.
InferEntityClassIdMixin	Infers entity class ids.
ImposeEntityClassIdMixin	Imposes entity class ids.
MakeRelationshipOnTheFlyMixin	Makes relationships on the fly.

Functions

_parse_csv_list(csv_list)	

spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.**_parse_csv_list**(*csv_list*)

class spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.**ConvertToDBMixin**

Base class for all mixins that convert model items (name-based) into database items (id-based).

build_lookup_dictionary(*db_map_data*)

Begins an operation to convert items.

_convert_to_db(*item*, *db_map*)

Returns a db item (id-based) from the given model item (name-based).

Parameters

- **item** (*dict*) – the model item
- **db_map** (*DiffDatabaseMapping*) – the database where the resulting item belongs

Returns
the db item list: error log

Return type
dict

class spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.**FillInAlternativeIdMixin**(**args*, ***kwargs*)

Bases: *ConvertToDBMixin*

Fills in alternative names.

Initializes lookup dicts.

build_lookup_dictionary(*db_map_data*)

Builds a name lookup dictionary for the given data.

Parameters
db_map_data (*dict*) – lists of model items keyed by DiffDatabaseMapping

_convert_to_db(*item*, *db_map*)

Returns a db item (id-based) from the given model item (name-based).

Parameters

- **item** (*dict*) – the model item
- **db_map** (*DiffDatabaseMapping*) – the database where the resulting item belongs

Returns
the db item list: error log

Return type
dict

class spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.**FillInParameterNameMixin**

Bases: *ConvertToDBMixin*

Fills in parameter names.

_convert_to_db(*item*, *db_map*)

Returns a db item (id-based) from the given model item (name-based).

Parameters

- **item** (*dict*) – the model item
- **db_map** (*DiffDatabaseMapping*) – the database where the resulting item belongs

Returns
the db item list: error log

Return type
dict

class spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.**FillInValueListIdMixin**(**args*, ***kwargs*)

Bases: *ConvertToDBMixin*

Fills in value list ids.

Initializes lookup dicts.

build_lookup_dictionary(*db_map_data*)

Builds a name lookup dictionary for the given data.

Parameters
db_map_data (*dict*) – lists of model items keyed by DiffDatabaseMapping

_convert_to_db(*item*, *db_map*)

Returns a db item (id-based) from the given model item (name-based).

Parameters

- **item** (*dict*) – the model item
- **db_map** (*DiffDatabaseMapping*) – the database where the resulting item belongs

Returns
the db item list: error log

Return type
dict

_fill_in_value_list_id(*item*, *db_map*)

Fills in the value list id in the given db item.

Parameters

- **item** (*dict*) – the db item
- **db_map** (*DiffDatabaseMapping*) – the database where the given item belongs

Returns
error log

Return type
list

class spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.**FillInEntityClassIdMixin**(**args*, ***kwargs*)

Bases: *ConvertToDBMixin*

Fills in entity_class ids.

Initializes lookup dicts.

build_lookup_dictionary(*db_map_data*)

Builds a name lookup dictionary for the given data.

Parameters
db_map_data (*dict*) – lists of model items keyed by DiffDatabaseMapping

_fill_in_entity_class_id(*item*, *db_map*)

Fills in the entity_class id in the given db item.

Parameters

- **item** (*dict*) – the db item
- **db_map** (*DiffDatabaseMapping*) – the database where the given item belongs

Returns
error log

Return type
list

_convert_to_db(*item*, *db_map*)

Returns a db item (id-based) from the given model item (name-based).

Parameters

- **item** (*dict*) – the model item
- **db_map** (*DiffDatabaseMapping*) – the database where the resulting item belongs

Returns
the db item list: error log

Return type
dict

class spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.**FillInEntityIdsMixin**(**args*, ***kwargs*)

Bases: *ConvertToDBMixin*

Fills in entity ids.

Initializes lookup dicts.

_add_entities_on_the_fly = False

build_lookup_dictionary(*db_map_data*)

Builds a name lookup dictionary for the given data.

Parameters
db_map_data (*dict*) – lists of model items keyed by DiffDatabaseMapping

_fill_in_entity_ids(*item*, *db_map*)

Fills in all possible entity ids keyed by entity_class id in the given db item (as there can be more than one entity for the same name).

Parameters

- **item** (*dict*) – the db item
- **db_map** (*DiffDatabaseMapping*) – the database where the given item belongs

Returns
error log

Return type
list

_convert_to_db(*item*, *db_map*)

Returns a db item (id-based) from the given model item (name-based).

Parameters

- **item** (*dict*) – the model item
- **db_map** (*DiffDatabaseMapping*) – the database where the resulting item belongs

Returns
the db item list: error log

Return type
dict

class spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.**FillInParameterDefinitionIdsMixin**(**args*, ***kwargs*)

Bases: *ConvertToDBMixin*

Fills in parameter_definition ids.

Initializes lookup dicts.

build_lookup_dictionary(*db_map_data*)

Builds a name lookup dictionary for the given data.

Parameters
db_map_data (*dict*) – lists of model items keyed by DiffDatabaseMapping

_fill_in_parameter_ids(*item*, *db_map*)

Fills in all possible parameter_definition ids keyed by entity_class id in the given db item (as there can be more than one parameter_definition for the same name).

Parameters

- **item** (*dict*) – the db item
- **db_map** (*DiffDatabaseMapping*) – the database where the given item belongs

Returns
error log

Return type
list

_convert_to_db(*item*, *db_map*)

Returns a db item (id-based) from the given model item (name-based).

Parameters

- **item** (*dict*) – the model item
- **db_map** (*DiffDatabaseMapping*) – the database where the resulting item belongs

Returns
the db item list: error log

Return type
dict

class spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.**InferEntityClassIdMixin**

Bases: *ConvertToDBMixin*

Infers entity class ids.

_convert_to_db(*item*, *db_map*)

Returns a db item (id-based) from the given model item (name-based).

Parameters

- **item** (*dict*) – the model item
- **db_map** (*DiffDatabaseMapping*) – the database where the resulting item belongs

Returns
the db item list: error log

Return type
dict

_infer_and_fill_in_entity_class_id(*item*, *db_map*)

Fills the entity_class id in the given db item, by intersecting entity ids and parameter ids. Then picks the correct entity id and parameter_definition id. Also sets the inferred entity_class name in the model.

Parameters

- **item** (*dict*) – the db item
- **db_map** (*DiffDatabaseMapping*) – the database where the given item belongs

Returns
error log

Return type
list

class spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.**ImposeEntityClassIdMixin**

Bases: *ConvertToDBMixin*

Imposes entity class ids.

_convert_to_db(*item*, *db_map*)

Returns a db item (id-based) from the given model item (name-based).

Parameters

- **item** (*dict*) – the model item
- **db_map** (*DiffDatabaseMapping*) – the database where the resulting item belongs

Returns
the db item list: error log

Return type
dict

_impose_entity_class_id(*item*, *db_map*)

Imposes the entity_class id from the model, to pick the correct entity id and parameter_definition id.

Parameters

- **item** (*dict*) – the db item
- **db_map** (*DiffDatabaseMapping*) – the database where the given item belongs

Returns
error log

Return type
list

class spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.**MakeRelationshipOnTheFlyMixin**(**args*, ***kwargs*)

Makes relationships on the fly.

Initializes lookup dicts.

static _make_unique_relationship_id(*item*)

Returns a unique name-based identifier for db relationships.

build_lookup_dictionaries(*db_map_data*)

Builds a name lookup dictionary for the given data.

Parameters
db_map_data (*dict*) – lists of model items keyed by DiffDatabaseMapping.

_make_relationship_on_the_fly(*item*, *db_map*)

Returns a database relationship item (id-based) from the given model parameter_value item (name-based).

Parameters

- **item** (*dict*) – the model parameter_value item
- **db_map** (*DiffDatabaseMapping*) – the database where the resulting item belongs

Returns
the db relationship item list: error log

Return type
dict

spinetoolbox.spine_db_editor.mvcmodels.parameter_value_list_item

Tree items for parameter_value lists.

authors

M. Marin (KTH)

date
28.6.2019

Module Contents

Classes

DBItem	An item representing a db.
ListItem	A list item.
ValueItem	Paints the item gray if it's the last.

class spinetoolbox.spine_db_editor.mvcmodels.parameter_value_list_item.**DBItem**(**args*, ***kwargs*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.EmptyChildMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.FetchMoreMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.StandardDBItem*

An item representing a db.

property item_type

property fetch_item_type

empty_child()

class spinetoolbox.spine_db_editor.mvcmodels.parameter_value_list_item.**ListItem**(*identifier=None*, *name=None*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.GrayIfLastMixin*,

spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.EditableMixin, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.EmptyChildMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.BoldTextMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.FetchMoreMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.LeafItem*

A list item.

Parameters
model (*MinimalTreeModel, optional*) – The model where the item belongs.

property item_type

_fetch_parents()

_make_item_data()

_do_finalize()

Do some final initialization after setting the parent.

empty_child()

data(*column*, *role=Qt.DisplayRole*)

Returns data for given column and role.

_make_item_to_add(*value*)

add_item_to_db(*db_item*)

update_item_in_db(*db_item*)

class spinetoolbox.spine_db_editor.mvcmodels.parameter_value_list_item.**ValueItem**(*identifier=None*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.GrayIfLastMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.EditableMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.LeafItem*

Paints the item gray if it's the last.

Parameters
model (*MinimalTreeModel, optional*) – The model where the item belongs.

property item_type

data(*column*, *role=Qt.DisplayRole*)

Returns data for given column and role.

list_index()

_make_item_to_add(*value*)

_make_item_to_update(*_column*, *value*)

add_item_to_db(*db_item*)

update_item_in_db(*db_item*)

spinetoolbox.spine_db_editor.mvcmodels.parameter_value_list_model

A tree model for parameter_value lists.

authors

M. Marin (KTH)

date

28.6.2019

Module Contents

Classes

ParameterValueListModel	A model to display parameter_value_list data in a tree view.

class spinetoolbox.spine_db_editor.mvcmodels.parameter_value_list_model.**ParameterValueListModel**(*parent*, *db_mngr*, **db_map*

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_model_base.TreeModelBase*

A model to display parameter_value_list data in a tree view.

Parameters

- **parent** (SpineDBEditor) –
- **db_mngr** (SpineDBManager) –
- ***db_maps** – DiffDatabaseMapping instances

add_parameter_value_lists(*db_map_data*)

add_list_values(*db_map_data*)

update_parameter_value_lists(*db_map_data*)

update_list_values(*db_map_data*)

remove_parameter_value_lists(*db_map_data*)

remove_list_values(*db_map_data*)

_items_per_list_item(*db_map_data*)

static _make_db_item(*db_map*)

static _top_children()

columnCount(*parent=QModelIndex()*)

Returns the number of columns under the given parent. Always 1.

index_name(*index*)

get_set_data_delayed(*index*)

Returns a function that ParameterValueEditor can call to set data for the given index at any later time, even if the model changes.

Parameters
index (*QModelIndex*) –

Returns
Callable

spinetoolbox.spine_db_editor.mvcmodels.pivot_model

Provides PivotModel.

author

P. Vennström (VTT)

date
1.11.2018

Module Contents

Classes

PivotModel

class spinetoolbox.spine_db_editor.mvcmodels.pivot_model.**PivotModel**

property rows

property columns

reset_model(*data*, *index_ids=()*, *rows=()*, *columns=()*, *frozen=()*, *frozen_value=()*)

Resets the model.

clear_model()

update_model(*data*)

add_to_model(*data*)

Adds data to model.

Parameters
data (*dict*) – pivot model data

Returns
added row count and added column count

Return type
tuple

remove_from_model(*data*)

_check_pivot(*rows*, *columns*, *frozen*, *frozen_value*)

Checks if given pivot is valid.

Returns
error message or None if no error

Return type
str, NoneType

_index_key_getter(*indexes*)

Returns an itemgetter that always returns tuples from list of indexes

Parameters
indexes (*tuple*) –

Returns
an itemgetter

Return type
Callable

_get_unique_index_values(*indexes*)

Returns unique indexes that match the frozen condition.

Parameters
indexes (*tuple*) – indexes to match

Returns
unique indexes

Return type
list

set_pivot(*rows*, *columns*, *frozen*, *frozen_value*)

Sets pivot.

set_frozen_value(*value*)

Sets values for the frozen indexes.

get_pivoted_data(*row_mask*, *column_mask*)

Returns data for indexes in row_mask and column_mask.

Parameters

- **row_mask** (*list*) –
- **column_mask** (*list*) –

Returns
list(list)

row_key(*row*)

column_key(*column*)

spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models

Provides pivot table models for the Tabular View.

author

P. Vennström (VTT)

date

1.11.2018

Module Contents

Classes

_ParameterFetchParent	
_EntityFetchParent	
_MemberObjectFetchParent	
PivotTableModelBase	**param parent**
TopLeftHeaderItem	Base class for all 'top left pivot headers'.
TopLeftObjectHeaderItem	A top left header for object_class.
TopLeftParameterHeaderItem	A top left header for parameter_definition.
TopLeftParameterIndexHeaderItem	A top left header for parameter index.
TopLeftAlternativeHeaderItem	A top left header for alternative.
TopLeftScenarioHeaderItem	A top left header for scenario.
TopLeftDatabaseHeaderItem	A top left header for database.
ParameterValuePivotTableModel	A model for the pivot table in parameter_value input type.
IndexExpansionPivotTableModel	A model for the pivot table in parameter index expansion input type.
RelationshipPivotTableModel	A model for the pivot table in relationship input type.
ScenarioAlternativePivotTableModel	A model for the pivot table in scenario alternative input type.
PivotTableSortFilterProxy	Initialize class.

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.**_ParameterFetchParent**(*fetch_item_type*, *parent*)

Bases: *spinetoolbox.helpers.ItemTypeFetchParent*

filter_query(*query*, *subquery*, *db_map*)

Filters the initial query created using the `fetch_item_type` property.

Parameters

- **query** (*Query*) – The query
- **subquery** (*Alias*) – The source of the query

- **db_map** (*DiffDatabaseMapping*) –

Returns
Query

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.**_EntityFetchParent**(*fetch_item_type*, *parent*)

Bases: *spinetoolbox.helpers.ItemTypeFetchParent*

filter_query(*query*, *subquery*, *db_map*)

Filters the initial query created using the `fetch_item_type` property.

Parameters

- **query** (*Query*) – The query
- **subquery** (*Alias*) – The source of the query
- **db_map** (*DiffDatabaseMapping*) –

Returns
Query

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.**_MemberObjectFetchParent**(*fetch_item_type*, *parent*)

Bases: *spinetoolbox.helpers.ItemTypeFetchParent*

filter_query(*query*, *subquery*, *db_map*)

Filters the initial query created using the `fetch_item_type` property.

Parameters

- **query** (*Query*) – The query
- **subquery** (*Alias*) – The source of the query
- **db_map** (*DiffDatabaseMapping*) –

Returns
Query

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.**PivotTableModelBase**(*parent*)

Bases: `PySide2.QtCore.QAbstractTableModel`

Parameters
parent (SpineDBEditor) –

abstract property item_type

Returns the item type.

property plot_x_column

Returns the index of the column designated as Y values for plotting or None.

_V_HEADER_WIDTH = 5

_MAX_FETCH_COUNT = 1000

_FETCH_DELAY = 0

model_data_changed

abstract _fetch_parents()

Yields fetch parents for this model.

Yields
FetchParent

_can_fetch_more_parent(*parent*)

canFetchMore(*_parent*)

_fetch_more_parent(*parent*)

fetchMore(*_parent*)

reset_data_count()

start_fetching()

fetch_more_rows()

fetch_more_columns()

abstract call_reset_model(*pivot=None*)

Parameters
pivot (*tuple, optional*) – list of rows, list of columns, list of frozen indexes, frozen value

abstract static make_delegate(*parent*)

reset_model(*data*, *index_ids*, *rows=()*, *columns=()*, *frozen=()*, *frozen_value=()*)

clear_model()

update_model(*data*)

Update model with new data, but doesn't grow the model.

Parameters
data (*dict*) –

add_to_model(*db_map_data*)

_emit_all_data_changed()

remove_from_model(*data*)

set_pivot(*rows*, *columns*, *frozen*, *frozen_value*)

set_frozen_value(*frozen_value*)

set_plot_x_column(*column*, *is_x*)

Sets or clears the X flag on a column

x_value(*index*)

Returns x value for given model index.

Parameters
index (*QModelIndex*) – model index

Returns
x value

Return type
Any

`x_parameter_name()`

Returns x column's parameter name.

Returns
parameter name

Return type
str

`headerRowCount()`

Returns number of rows occupied by header.

`headerColumnCount()`

Returns number of columns occupied by header.

`dataRowCount()`

Returns number of rows that contain actual data.

`dataColumnCount()`

Returns number of columns that contain actual data.

`emptyRowCount()`

`emptyColumnCount()`

`rowCount(parent=QModelIndex())`

Number of rows in table, number of header rows + datarows + 1 empty row

`columnCount(parent=QModelIndex())`

Number of columns in table, number of header columns + datacolumns + 1 empty columns

`flags(index)`

Roles for data

`top_left_indexes()`

Returns indexes in the top left area.

Returns
list(QModelIndex): top indexes (horizontal headers, associated to rows) list(QModelIndex): left indexes (vertical headers, associated to columns)

`index_within_top_left(index)`

`index_in_top(index)`

`index_in_left(index)`

`index_in_top_left(index)`

Returns whether or not the given index is in top left corner, where pivot names are displayed

`index_in_column_headers(index)`

Returns whether or not the given index is in column headers (horizontal) area

`index_in_row_headers(index)`

Returns whether or not the given index is in row headers (vertical) area

`index_in_headers(index)`

index_in_empty_column_headers(*index*)

Returns whether or not the given index is in empty column headers (vertical) area

index_in_empty_row_headers(*index*)

Returns whether or not the given index is in empty row headers (vertical) area

index_in_data(*index*)

Returns whether or not the given index is in data area

column_is_index_column(*column*)

Returns True if column is the column containing expanded parameter_value indexes.

headerData(*section*, *orientation*, *role=Qt.DisplayRole*)

map_to_pivot(*index*)

Returns a tuple of row and column in the pivot model that corresponds to the given model index.

Parameters
index (*QModelIndex*) –

Returns
row int: column

Return type
int

top_left_id(*index*)

Returns the id of the top left header corresponding to the given header index.

Parameters
index (*QModelIndex*) –

Returns
int, NoneType

_header_id(*index*)

Returns the id of the given row or column header index.

Parameters
index (*QModelIndex*) –

Returns
int, NoneType

_header_ids(*row*, *column*)

Returns the ids for the headers at given row *and* column.

Parameters

- **row** (*int*) –
- **column** (*int*) –

Returns
tuple(int)

header_name(*index*)

Returns the name corresponding to the given header index. Used by PivotTableView.

Parameters
index (*QModelIndex*) –

Returns
str

_color_data(*index*)

_text_alignment_data(*index*)

_header_data(*index*, *role=Qt.DisplayRole*)

_header_name(*top_left_id*, *header_id*)

abstract _data(*index*, *role*)

data(*index*, *role=Qt.DisplayRole*)

setData(*index*, *value*, *role=Qt.EditRole*)

batch_set_data(*indexes*, *values*)

_batch_set_inner_data(*inner_data*)

abstract _do_batch_set_inner_data(*row_map*, *column_map*, *data*, *values*)

_batch_set_header_data(*header_data*)

_batch_set_empty_header_data(*header_data*, *get_top_left_id*)

receive_data_added_or_removed(*db_map_data*, *action*)

receive_objects_added_or_removed(*db_map_data*, *action*)

receive_relationships_added_or_removed(*db_map_data*, *action*)

receive_parameter_definitions_added_or_removed(*db_map_data*, *action*)

receive_alternatives_added_or_removed(*db_map_data*, *action*)

receive_parameter_values_added_or_removed(*db_map_data*, *action*)

receive_scenarios_added_or_removed(*db_map_data*, *action*)

receive_scenarios_updated(*db_map_data*)

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.**TopLeftHeaderItem**(*model*)

Base class for all 'top left pivot headers'. Represents a header located in the top left area of the pivot table.

Parameters
model (PivotTableModelBase) –

property model

property db_mngr

_get_header_data_from_db(*item_type*, *header_id*, *field_name*, *role*)

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.**TopLeftObjectHeaderItem**(*model*, *class_name*, *class_id*)

Bases: *TopLeftHeaderItem*

A top left header for object_class.

Parameters
model (PivotTableModelBase) –

property header_type

property name

header_data(*header_id*, *role=Qt.DisplayRole*)

update_data(*db_map_data*)

add_data(*names*)

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.**TopLeftParameterHeaderItem**(*model*)

Bases: *TopLeftHeaderItem*

A top left header for parameter_definition.

Parameters
model (PivotTableModelBase) –

property header_type

property name

header_data(*header_id*, *role=Qt.DisplayRole*)

update_data(*db_map_data*)

add_data(*names*)

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.**TopLeftParameterIndexHeaderItem**(*model*)

Bases: *TopLeftHeaderItem*

A top left header for parameter index.

Parameters
model (PivotTableModelBase) –

property header_type

property name

header_data(*header_id*, *role=Qt.DisplayRole*)

update_data(*db_map_data*)

add_data(*_names*)

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.**TopLeftAlternativeHeaderItem**(*model*)

Bases: *TopLeftHeaderItem*

A top left header for alternative.

Parameters
model (PivotTableModelBase) –

property header_type

property name

header_data(*header_id*, *role=Qt.DisplayRole*)

update_data(*db_map_data*)

add_data(*names*)

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.**TopLeftScenarioHeaderItem**(*model*)

Bases: *TopLeftHeaderItem*

A top left header for scenario.

Parameters
model (PivotTableModelBase) –

property header_type

property name

header_data(*header_id*, *role=Qt.DisplayRole*)

update_data(*db_map_data*)

add_data(*names*)

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.**TopLeftDatabaseHeaderItem**(*model*)

Bases: *TopLeftHeaderItem*

A top left header for database.

Parameters
model (PivotTableModelBase) –

property header_type

property name

header_data(*header_id*, *role=Qt.DisplayRole*)

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.**ParameterValuePivotTableModel**(*parent*)

Bases: *PivotTableModelBase*

A model for the pivot table in parameter_value input type.

Parameters
parent (SpineDBEditor) –

property item_type

Returns the item type.

_fetch_parents()

Yields fetch parents for this model.

Yields
FetchParent

db_map_object_ids(*index*)

Returns db_map and object ids for given index. Used by PivotTableView.

Returns
DatabaseMapping, list

_db_map_object_ids(*header_ids*)

all_header_names(*index*)

Returns the object, parameter, alternative, and db names corresponding to the given data index.

Parameters
index (*QModelIndex*) –

Returns
object names str: parameter name str: alternative name str: db name

Return type
list(str)

index_name(*index*)

Returns a string that concatenates the object and parameter names corresponding to the given data index. Used by plotting and ParameterValueEditor.

Parameters
index (*QModelIndex*) –

Returns
str

column_name(*column*)

Returns a string that concatenates the object and parameter names corresponding to the given column. Used by plotting.

Parameters
column (*int*) –

Returns
str

call_reset_model(*pivot=None*)

See base class.

static make_delegate(*parent*)

_default_pivot(*data*)

_data(*index*, *role*)

_do_batch_set_inner_data(*row_map*, *column_map*, *data*, *values*)

_object_parameter_value_to_add(*db_map*, *header_ids*, *value_and_type*)

_relationship_parameter_value_to_add(*db_map*, *header_ids*, *value_and_type*, *rel_id_lookup*)

_make_parameter_value_to_add()

static _parameter_value_to_update(*id_*, *header_ids*, *value_and_type*)

_batch_set_parameter_value_data(*row_map*, *column_map*, *data*, *values*)

Sets parameter values in batch.

_add_parameter_values(*db_map_data*)

_update_parameter_values(*db_map_data*)

get_set_data_delayed(*index*)

Returns a function that ParameterValueEditor can call to set data for the given index at any later time, even if the model changes.

Parameters
index (*QModelIndex*) –

Returns
function

receive_objects_added_or_removed(*db_map_data*, *action*)

receive_relationships_added_or_removed(*db_map_data*, *action*)

receive_parameter_definitions_added_or_removed(*db_map_data*, *action*)

receive_alternatives_added_or_removed(*db_map_data*, *action*)

receive_parameter_values_added_or_removed(*db_map_data*, *action*)

_load_empty_parameter_value_data(**args*, ***kwargs*)

_load_full_parameter_value_data(**args*, ***kwargs*)

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.**IndexExpansionPivotTableModel**(*parent*)

Bases: *ParameterValuePivotTableModel*

A model for the pivot table in parameter index expansion input type.

Parameters
parent (SpineDBEditor) –

call_reset_model(*pivot=None*)

See base class.

flags(*index*)

Roles for data

column_is_index_column(*column*)

Returns True if column is the column containing expanded parameter_value indexes.

_load_empty_parameter_value_data(**args*, ***kwargs*)

_load_full_parameter_value_data(**args*, ***kwargs*)

_data(*index*, *role*)

static _parameter_value_to_update(*id_*, *header_ids*, *value_and_type*)

_update_parameter_values(*db_map_data*)

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.**RelationshipPivotTableModel**(*parent*)

Bases: *PivotTableModelBase*

A model for the pivot table in relationship input type.

Parameters
parent (SpineDBEditor) –

property item_type

Returns the item type.

_fetch_parents()

Yields fetch parents for this model.

Yields
FetchParent

call_reset_model(*pivot=None*)

See base class.

static make_delegate(*parent*)

_default_pivot(*data*)

_data(*index*, *role*)

_do_batch_set_inner_data(*row_map*, *column_map*, *data*, *values*)

_batch_set_relationship_data(*row_map*, *column_map*, *data*, *values*)

receive_objects_added_or_removed(*db_map_data*, *action*)

receive_relationships_added_or_removed(*db_map_data*, *action*)

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.**ScenarioAlternativePivotTableModel**(*paren*

Bases: *PivotTableModelBase*

A model for the pivot table in scenario alternative input type.

Parameters
parent (SpineDBEditor) –

property item_type

Returns the item type.

_fetch_parents()

Yields fetch parents for this model.

Yields
FetchParent

call_reset_model(*pivot=None*)

See base class.

static make_delegate(*parent*)

_default_pivot(*data*)

_data(*index*, *role*)

_do_batch_set_inner_data(*row_map*, *column_map*, *data*, *values*)

_batch_set_scenario_alternative_data(*row_map*, *column_map*, *data*, *values*)

receive_scenarios_updated(*db_map_data*)

receive_alternatives_added_or_removed(*db_map_data*, *action*)

receive_scenarios_added_or_removed(*db_map_data*, *action*)

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.**PivotTableSortFilterProxy**(*parent=None*)

Bases: PySide2.QtCore.QSortFilterProxyModel

Initialize class.

model_data_changed

setSourceModel(*model*)

set_filter(*identifier*, *filter_value*)

Sets filter for a given index (object_class) name.

Parameters

- **identifier** (*int*) – index identifier
- **filter_value** (*set, None*) – A set of accepted values, or None if no filter (all pass)

clear_filter()

accept_index(*index*, *index_ids*)

filterAcceptsRow(*source_row*, *source_parent*)

Returns true if the item in the row indicated by the given source_row and source_parent should be included in the model; otherwise returns false.

filterAcceptsColumn(*source_column*, *source_parent*)

Returns true if the item in the column indicated by the given source_column and source_parent should be included in the model; otherwise returns false.

batch_set_data(*indexes*, *values*)

spinetoolbox.spine_db_editor.mvcmodels.single_parameter_models

Single models for parameter definitions and values (as 'for a single entity').

authors

M. Marin (KTH)

date

28.6.2019

Module Contents

Classes

HalfSortedTableModel	Table model for outlining simple tabular data.
SingleParameterModel	A parameter model for a single entity_class to go in a CompoundParameterModel.
SingleObjectParameterMixin	Associates a parameter model with a single object_class.
SingleRelationshipParameterMixin	Associates a parameter model with a single relationship_class.
SingleParameterDefinitionMixin	A parameter_definition model for a single entity_class.
SingleParameterValueMixin	A parameter_value model for a single entity_class.
SingleObjectParameterDefinitionModel	An object parameter_definition model for a single object_class.
SingleRelationshipParameterDefinitionModel	A relationship parameter_definition model for a single relationship_class.
SingleObjectParameterValueModel	An object parameter_value model for a single object_class.
SingleRelationshipParameterValueModel	A relationship parameter_value model for a single relationship_class.

class spinetoolbox.spine_db_editor.mvcmodels.single_parameter_models.**HalfSortedTableModel**(*parent=None*, *header=None*, *lazy=True*)

Bases: *spinetoolbox.mvcmodels.minimal_table_model.MinimalTableModel*

Table model for outlining simple tabular data.

Parameters

- **parent** (*QObject, optional*) – the parent object
- **header** (*list of str*) – header labels
- **lazy** (*boolean*) – if True, fetches data lazily

reset_model(*main_data=None*)

Reset model.

add_rows(*data*)

_sort_key(*element*)

class spinetoolbox.spine_db_editor.mvcmodels.single_parameter_models.**SingleParameterModel**(*header*, *db_mngr*, *db_map*, *entity_class_id*, *committed*, *lazy=False*)

Bases: *HalfSortedTableModel*

A parameter model for a single entity_class to go in a CompoundParameterModel. Provides methods to associate the model to an entity_class as well as to filter entities within the class.

Init class.

Parameters
header (*list*) – list of field names for the header

abstract property item_type

The item type, either 'parameter_value' or 'parameter_definition', required by the data method.

abstract property entity_class_type

The entity_class type, either 'object_class' or 'relationship_class'.

property entity_class_name_field

property entity_class_name

property entity_class_id_key

property value_field

property fixed_fields

property group_fields

property parameter_definition_id_key

property can_be_filtered

__lt__(*other*)

item_id(*row*)

Returns parameter id for row.

Parameters
row (*int*) – row index

Returns
parameter id

Return type
int

item_ids()

Returns model's parameter ids.

Returns
ids

Return type
set of int

db_item(*index*)

_db_item(*row*)

db_item_from_id(*id_*)

db_items()

flags(*index*)

Make fixed indexes non-editable.

get_field_item_data(*field*)

Returns item data for given field.

Parameters
field (*str*) – A field from the header

Returns
str, str

get_id_key(*field*)

get_field_item(*field*, *db_item*)

Returns a db item corresponding to the given field from the table header, or an empty dict if the field doesn't contain db items.

data(*index*, *role=Qt.DisplayRole*)

Gets the id and database for the row, and reads data from the db manager using the item_type property. Paint the object_class icon next to the name. Also paint background of fixed indexes gray and apply custom format to JSON fields.

batch_set_data(*indexes*, *data*)

Sets data for indexes in batch. Sets data directly in database using db mngr. If successful, updated data will be automatically seen by the data method.

abstract update_items_in_db(*items*)

Update items in db. Required by batch_set_data

_filter_accepts_row(*row*)

filter_accepts_item(*item*)

set_auto_filter(*field*, *values*)

_auto_filter_accepts_item(*item*)

Returns the result of the auto filter.

accepted_rows()

Yields accepted rows, for convenience.

_get_field_item(*field*, *id_*)

Returns a item from the db_mngr.get_item depending on the field. If a field doesn't correspond to a item in the database then an empty dict is returned.

class spinetoolbox.spine_db_editor.mvcmodels.single_parameter_models.**SingleObjectParameterMixin**

Associates a parameter model with a single object_class.

property entity_class_type

class spinetoolbox.spine_db_editor.mvcmodels.single_parameter_models.**SingleRelationshipParameterMixin**

Associates a parameter model with a single relationship_class.

property entity_class_type

class spinetoolbox.spine_db_editor.mvcmodels.single_parameter_models.**SingleParameterDefinitionMixin**(**arg*, ***kw*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.FillInParameterNameMixin*, *spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.FillInValueListIdMixin*

A parameter_definition model for a single entity_class.

Initializes lookup dicts.

property item_type

_sort_key(*element*)

update_items_in_db(*items*)

Update items in db.

Parameters
item (*list*) – dictionary-items

class spinetoolbox.spine_db_editor.mvcmodels.single_parameter_models.**SingleParameterValueMixin**(**args*, ***kwargs*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.FillInAlternativeIdMixin*, *spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.ImposeEntityClassIdMixin*, *spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.FillInParameterDefinitionIdsMixin*, *spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.FillInEntityIdsMixin*

A parameter_value model for a single entity_class.

Initializes lookup dicts.

property item_type

abstract property entity_type

Either 'object' or "relationship'.

property entity_id_key

property entity_name_key

property entity_name_key_in_cache

_filter_db_map_class_entity_ids

_filter_alternative_ids

_filter_entity_ids

_sort_key(*element*)

set_filter_entity_ids(*db_map_class_entity_ids*)

set_filter_alternative_ids(*db_map_alternative_ids*)

filter_accepts_item(*item*)

Reimplemented to also account for the entity and alternative filter.

_entity_filter_accepts_item(*item*)

Returns the result of the entity filter.

_alternative_filter_accepts_item(*item*)

Returns the result of the alternative filter.

update_items_in_db(*items*)

Update items in db.

Parameters
items (*list*) – dictionary-items

class spinetoolbox.spine_db_editor.mvcmodels.single_parameter_models.**SingleObjectParameterDefinitionMod**

Bases: *SingleObjectParameterMixin*, *SingleParameterDefinitionMixin*, *SingleParameterModel*

An object parameter_definition model for a single object_class.

Initializes lookup dicts.

class spinetoolbox.spine_db_editor.mvcmodels.single_parameter_models.**SingleRelationshipParameterDefinit**

Bases: *SingleRelationshipParameterMixin*, *SingleParameterDefinitionMixin*, *SingleParameterModel*

A relationship parameter_definition model for a single relationship_class.

Initializes lookup dicts.

class spinetoolbox.spine_db_editor.mvcmodels.single_parameter_models.**SingleObjectParameterValueModel**(**a*
***p*

Bases: *SingleObjectParameterMixin*, *SingleParameterValueMixin*, *SingleParameterModel*

An object parameter_value model for a single object_class.

Initializes lookup dicts.

property entity_type

Either 'object' or "relationship'.

class spinetoolbox.spine_db_editor.mvcmodels.single_parameter_models.**SingleRelationshipParameterValueMo**

Bases: *SingleRelationshipParameterMixin*, *spinetoolbox.spine_db_editor.mvcmodels.parameter_mixins.MakeRelationshipOnTheFlyMixin*, *SingleParameterValueMixin*, *SingleParameterModel*

A relationship parameter_value model for a single relationship_class.

Initializes lookup dicts.

property entity_type

Either 'object' or "relationship'.

update_items_in_db(*items*)

Update items in db.

Parameters
item (*list*) – dictionary-items

spinetoolbox.spine_db_editor.mvcmodels.tool_feature_item

Classes to represent tool and feature items in a tree.

authors

M. Marin (KTH)

date

1.9.2020

Module Contents

Classes

FeatureRootItem	A feature root item.
ToolRootItem	A tool root item.
FeatureLeafItem	A feature leaf item.
ToolLeafItem	A tool leaf item.
ToolFeatureRootItem	A tool_feature root item.
ToolFeatureLeafItem	A tool feature leaf item.
ToolFeatureRequiredItem	A tool feature required item.
ToolFeatureMethodRootItem	A tool_feature_method root item.
ToolFeatureMethodLeafItem	A tool_feature_method leaf item.

Attributes

_FEATURE_ICON
_TOOL_ICON
_METHOD_ICON

spinetoolbox.spine_db_editor.mvcmodels.tool_feature_item.**_FEATURE_ICON** =

spinetoolbox.spine_db_editor.mvcmodels.tool_feature_item.**_TOOL_ICON** =

spinetoolbox.spine_db_editor.mvcmodels.tool_feature_item.**_METHOD_ICON** =

class spinetoolbox.spine_db_editor.mvcmodels.tool_feature_item.**FeatureRootItem**(**args*, ***kwargs*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.EmptyChildRootItem*

A feature root item.

property item_type

property display_data

property icon_code

empty_child()

class spinetoolbox.spine_db_editor.mvcmodels.tool_feature_item.**ToolRootItem**(**args*, ***kwargs*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.EmptyChildRootItem*

A tool root item.

property item_type

property display_data

property icon_code

empty_child()

class spinetoolbox.spine_db_editor.mvcmodels.tool_feature_item.**FeatureLeafItem**(*identifier=None*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.GrayIfLastMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.EditableMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.LeafItem*

A feature leaf item.

Parameters
model (*MinimalTreeModel*, *optional*) – The model where the item belongs.

property item_type

property item_data

property tool_tip

_make_item_data()

add_item_to_db(*db_item*)

update_item_in_db(*db_item*)

flags(*column*)

Makes items editable.

_make_item_to_add(*value*)

_make_item_to_update(*column*, *value*)

_get_ids_from_feat_name(*feature_name*)

class spinetoolbox.spine_db_editor.mvcmodels.tool_feature_item.**ToolLeafItem**(*identifier=None*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.GrayIfLastMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.EditableMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.LeafItem*

A tool leaf item.

Parameters
model (*MinimalTreeModel*, *optional*) – The model where the item belongs.

property item_type

add_item_to_db(*db_item*)

update_item_in_db(*db_item*)

_do_finalize()

Do some final initialization after setting the parent.

class spinetoolbox.spine_db_editor.mvcmodels.tool_feature_item.**ToolFeatureRootItem**(**args*, ***kwargs*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.EmptyChildRootItem*

A tool_feature root item.

property item_type

property display_data

property tool_tip

property icon_code

property feature_id_list

flags(*column*)

Enables the item and makes it selectable.

empty_child()

class spinetoolbox.spine_db_editor.mvcmodels.tool_feature_item.**ToolFeatureLeafItem**(*identifier=None*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.GrayIfLastMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.LeafItem*

A tool feature leaf item.

Parameters

model (*MinimalTreeModel, optional*) – The model where the item belongs.

property item_type

property item_data

_do_finalize()

Do some final initialization after setting the parent.

_make_item_to_add(*value*)

add_item_to_db(*db_item*)

update_item_in_db(*db_item*)

flags(*column*)

Enables the item and makes it selectable.

class spinetoolbox.spine_db_editor.mvcmodels.tool_feature_item.**ToolFeatureRequiredItem**(*model=None*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.StandardTreeItem*

A tool feature required item.

Parameters

model (*MinimalTreeModel, optional*) – The model where the item belongs.

property item_type

flags(*column*)

Enables the item and makes it selectable.

data(*column*, *role=Qt.DisplayRole*)

Returns data for given column and role.

set_data(*column*, *value*, *role=Qt.EditRole*)

Sets data for this item.

Parameters

- **column** (*int*) – column index
- **value** (*object*) – a new value
- **role** (*int*) – role of the new value

Returns

True if data was set successfully, False otherwise

Return type

bool

has_children()

Returns whether or not this item has or could have children.

class spinetoolbox.spine_db_editor.mvcmodels.tool_feature_item.**ToolFeatureMethodRootItem**(**args*, ***kwargs*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.EmptyChildRootItem*

A tool_feature_method root item.

Parameters

model (*MinimalTreeModel, optional*) – The model where the item belongs.

property item_type

property display_data

property icon_code

_do_finalize()

Do some final initialization after setting the parent.

_fetch_parents()

empty_child()

class spinetoolbox.spine_db_editor.mvcmodels.tool_feature_item.**ToolFeatureMethodLeafItem**(*identifier=None*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.GrayIfLastMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.LeafItem*

A tool_feature_method leaf item.

Parameters

model (*MinimalTreeModel, optional*) – The model where the item belongs.

property item_type

property tool_feature_item

property item_data

_make_item_data()

flags(*column*)

Enables the item and makes it selectable.

_make_item_to_add(*value*)

_make_item_to_update(*column*, *value*)

_get_method_index(*parameter_value_list_id*, *method*)

add_item_to_db(*db_item*)

update_item_in_db(*db_item*)

spinetoolbox.spine_db_editor.mvcmodels.tool_feature_model

Models to represent tools and features in a tree.

authors

M. Marin (KTH)

date

1.0.2020

Module Contents

Classes

ToolFeatureModel	A model to display tools and features in a tree view.

class spinetoolbox.spine_db_editor.mvcmodels.tool_feature_model.**ToolFeatureModel**(*parent*, *db_mngr*, **db_maps*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_model_base.TreeModelBase*

A model to display tools and features in a tree view.

Parameters

- **parent** (SpineDBEditor) –
- **db_mngr** (SpineDBManager) –
- **db_maps** (*iter*) – DiffDatabaseMapping instances

Initialize class

static _make_db_item(*db_map*)

static _top_children()

static make_feature_name(*entity_class_name*, *parameter_definition_name*)

_begin_set_features(*db_map*)

get_all_feature_names(*db_map*)

get_feature_data(*db_map*, *feature_name*)

_begin_set_feature_method(*db_map*, *parameter_value_list_id*)

get_all_feature_methods(*db_map*, *parameter_value_list_id*)

get_method_index(*db_map*, *parameter_value_list_id*, *method*)

_tools_per_root(*db_map_data*)

_features_per_root(*db_map_data*)

_tool_features_per_root(*db_map_data*)

_tool_feature_methods_per_root(*db_map_data*)

add_features(*db_map_data*)

add_tools(*db_map_data*)

add_tool_features(*db_map_data*)

add_tool_feature_methods(*db_map_data*)

update_features(*db_map_data*)

update_tools(*db_map_data*)

update_tool_features(*db_map_data*)

update_tool_feature_methods(*db_map_data*)

remove_features(*db_map_data*)

remove_tools(*db_map_data*)

remove_tool_features(*db_map_data*)

remove_tool_feature_methods(*db_map_data*)

supportedDropActions()

mimeData(*indexes*)

Builds a dict mapping db name to item type to a list of ids.

Returns
QMimeData

canDropMimeData(*data*, *drop_action*, *row*, *column*, *parent*)

dropMimeData(*data*, *drop_action*, *row*, *column*, *parent*)

spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility

A tree model for parameter_value lists.

authors

M. Marin (KTH)

date

28.6.2019

Module Contents

Classes

StandardTreeItem	A tree item that fetches their children as they are inserted.
EditableMixin	
GrayIfLastMixin	Paints the item gray if it's the last.
BoldTextMixin	Bolds text.
EmptyChildMixin	Guarantees there's always an empty child.
SortsChildrenMixin	
FetchMoreMixin	
StandardDBItem	An item representing a db.
RootItem	A root item.
EmptyChildRootItem	Guarantees there's always an empty child.
LeafItem	A tree item that fetches their children as they are inserted.
ListValueFetchParent	

class spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.**StandardTreeItem**(*model=None*)

Bases: *spinetoolbox.mvcmodels.minimal_tree_model.TreeItem*

A tree item that fetches their children as they are inserted.

Parameters

model (*MinimalTreeModel, optional*) – The model where the item belongs.

property item_type

property db_mngr

property display_data

property icon_code

property tool_tip

property display_icon

property non_empty_children

property children_ids

data(*column*, *role=Qt.DisplayRole*)

Returns data for given column and role.

set_data(*column*, *value*, *role=Qt.DisplayRole*)

Sets data for this item.

Parameters

- **column** (*int*) – column index
- **value** (*object*) – a new value
- **role** (*int*) – role of the new value

Returns

True if data was set successfully, False otherwise

Return type

bool

class spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.**EditableMixin**

flags(*column*)

Makes items editable.

class spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.**GrayIfLastMixin**

Paints the item gray if it's the last.

data(*column*, *role=Qt.DisplayRole*)

class spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.**BoldTextMixin**

Bolds text.

data(*column*, *role=Qt.DisplayRole*)

class spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.**EmptyChildMixin**

Guarantees there's always an empty child.

property non_empty_children

abstract empty_child()

_do_finalize()

class spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.**SortsChildrenMixin**

insert_children_sorted(*children*)

class spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.**FetchMoreMixin**(**args*, ***kwargs*)

property fetch_item_type

_fetch_parents()

can_fetch_more()

fetch_more()

class spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.**StandardDBItem**(*db_map*)

Bases: *SortsChildrenMixin*, *StandardTreeItem*

An item representing a db.

Init class.

Args

db_mngr (SpineDBManager) db_map (DiffDatabaseMapping)

property item_type

data(*column*, *role=Qt.DisplayRole*)

Shows Spine icon for fun.

class spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.**RootItem**(**args*, ***kwargs*)

Bases: *SortsChildrenMixin*, *BoldTextMixin*, *FetchMoreMixin*, *StandardTreeItem*

A root item.

abstract property item_type

property db_map

class spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.**EmptyChildRootItem**(**args*, ***kwargs*)

Bases: *EmptyChildMixin*, *RootItem*

Guarantees there's always an empty child.

abstract empty_child()

class spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.**LeafItem**(*identifier=None*)

Bases: *StandardTreeItem*

A tree item that fetches their children as they are inserted.

Parameters

model (*MinimalTreeModel, optional*) – The model where the item belongs.

abstract property item_type

property db_map

property id

property item_data

property name

_make_item_data()

abstract add_item_to_db(*db_item*)

abstract update_item_in_db(*db_item*)

header_data(*column*)

data(*column*, *role=Qt.DisplayRole*)

Returns data for given column and role.

set_data(*column*, *value*, *role=Qt.EditRole*)

Sets data for this item.

Parameters

- **column** (*int*) – column index
- **value** (*object*) – a new value
- **role** (*int*) – role of the new value

Returns
True if data was set successfully, False otherwise

Return type
bool

_make_item_to_add(*value*)

_make_item_to_update(*column*, *value*)

handle_updated_in_db()

can_fetch_more()

Returns whether or not this item can fetch more.

class spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.**ListValueFetchParent**(*parameter_value_list_id*)

Bases: *spinetoolbox.helpers.FetchParent*

property fetch_item_type

Returns the type of item to fetch, e.g., "object_class". Used to create an initial query for this item.

Returns
str

filter_query(*query*, *subquery*, *db_map*)

Filters the initial query created using the `fetch_item_type` property.

Parameters

- **query** (*Query*) – The query
- **subquery** (*Alias*) – The source of the query
- **db_map** (*DiffDatabaseMapping*) –

Returns
Query

spinetoolbox.spine_db_editor.mvcmodels.tree_model_base

Models to represent things in a tree.

authors

M. Marin (KTH)

date
1.0.2020

Module Contents

Classes

TreeModelBase	A base model to display items in a tree view.

class spinetoolbox.spine_db_editor.mvcmodels.tree_model_base.**TreeModelBase**(*parent*, *db_mngr*, **db_maps*)

Bases: *spinetoolbox.mvcmodels.minimal_tree_model.MinimalTreeModel*

A base model to display items in a tree view.

Parameters

- **parent** (*SpineDBEditor*) –
- **db_mngr** (*SpineDBManager*) –
- ***db_maps** – DiffDatabaseMapping instances

columnCount(*parent=QModelIndex()*)

Returns the number of columns under the given parent. Always 2.

Returns
column count

Return type
int

headerData(*section*, *orientation*, *role=Qt.DisplayRole*)

build_tree()

Builds tree.

abstract static _make_db_item(*db_map*)

abstract static _top_children()

_items_per_db_item(*db_map_data*)

_items_per_root(*db_map_data*, *root_number=0*)

static _db_map_data_per_id(*db_map_data*, *id_key*)

_update_leaf_items(*root_item*, *ids*)

static _remove_leaf_items(*root_item*, *ids*)

static _insert_items(*parent_item*, *db_items*, *make_child*)

Inserts items at right positions. Items with commit_id are kept sorted. Items without a commit_id are put at the end.

Parameters

- **parent_item** (*TreeItem*) –
- **db_items** (*list of dict*) – database items
- **make_child** (*Callable*) – A function that receives an integer id and returns a TreeItem

static **db_item**(*item*)

db_row(*item*)

spinetoolbox.spine_db_editor.ui

Automatically generated UI modules for Spine db editor.

authors

M. Marin (KTH)

date

13.5.2020

Submodules

spinetoolbox.spine_db_editor.ui.scenario_generator

Module Contents

Classes

Ui_Form

class spinetoolbox.spine_db_editor.ui.scenario_generator.**Ui_Form**

Bases: object

setupUi(*Form*)

retranslateUi(*Form*)

spinetoolbox.spine_db_editor.ui.select_database_items_dialog

Module Contents

Classes

Ui_Dialog

class spinetoolbox.spine_db_editor.ui.select_database_items_dialog.**Ui_Dialog**

Bases: object

setupUi(*Dialog*)

retranslateUi(*Dialog*)

spinetoolbox.spine_db_editor.ui.spine_db_editor_window

Module Contents

Classes

Ui_MainWindow

class spinetoolbox.spine_db_editor.ui.spine_db_editor_window.**Ui_MainWindow**

Bases: object

setupUi(*MainWindow*)

retranslateUi(*MainWindow*)

spinetoolbox.spine_db_editor.widgets

Interface logic for Spine db editor.

authors

M. Marin (KTH)

date

13.5.2020

Submodules

spinetoolbox.spine_db_editor.widgets.add_items_dialogs

Classes for custom QDialogs to add items to databases.

author

M. Marin (KTH)

date

13.5.2018

Module Contents

Classes

AddReadyRelationshipsDialog	A dialog to let the user add new 'ready' relationships.
AddItemsDialog	A dialog to query user's preferences for new db items.
AddObjectClassesDialog	A dialog to query user's preferences for new object classes.
AddObjectsDialog	A dialog to query user's preferences for new objects.
AddRelationshipClassesDialog	A dialog to query user's preferences for new relationship classes.
AddOrManageRelationshipsDialog	A dialog to query user's preferences for new relationships.
AddRelationshipsDialog	A dialog to query user's preferences for new relationships.
ManageRelationshipsDialog	A dialog to query user's preferences for managing relationships.
ObjectGroupDialogBase	**param parent** data store widget
AddObjectGroupDialog	**param parent** data store widget
ManageMembersDialog	**param parent** data store widget

class spinetoolbox.spine_db_editor.widgets.add_items_dialogs.**AddReadyRelationshipsDialog**(*parent*, *relationships_class*, *relationships*, *db_mngr*, **db_maps*)

Bases: *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.ManageItemsDialogBase*

A dialog to let the user add new 'ready' relationships.

Parameters

- **parent** (SpineDBEditor) –
- **relationships_class** (*dict*) –
- **relationships** (*list(list(str))*) –
- **db_mngr** (SpineDBManager) –
- ***db_maps** – DiffDatabaseMapping instances

make_table_view()

populate_table_view()

connect_signals()

Connect signals to slots.

_handle_table_view_cell_clicked(*row*, *column*)

_handle_table_view_current_changed(*current*, *_previous*)

accept()

class spinetoolbox.spine_db_editor.widgets.add_items_dialogs.**AddItemsDialog**(*parent*, *db_mngr*, **db_maps*)

Bases: *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.ManageItemsDialog*

A dialog to query user's preferences for new db items.

Parameters

- **parent** (SpineDBEditor) –
- **db_mngr** (SpineDBManager) –
- ***db_maps** – DiffDatabaseMapping instances

connect_signals()

Connect signals to slots.

remove_selected_rows(*checked=True*)

all_databases(*row*)

Returns a list of db names available for a given row. Used by delegates.

class spinetoolbox.spine_db_editor.widgets.add_items_dialogs.**AddObjectClassesDialog**(*parent*, *db_mngr*, **db_maps*)

Bases: *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.ShowIconColorEditorMixin*, *AddItemsDialog*

A dialog to query user's preferences for new object classes.

Parameters

- **parent** (SpineDBEditor) –
- **db_mngr** (SpineDBManager) –
- ***db_maps** – DiffDatabaseMapping instances

connect_signals()

Connect signals to slots.

all_db_maps(*row*)

Returns a list of db maps available for a given row. Used by ShowIconColorEditorMixin.

accept()

Collect info from dialog and try to add items.

class spinetoolbox.spine_db_editor.widgets.add_items_dialogs.**AddObjectsDialog**(*parent*, *parent_item*, *db_mngr*, **db_maps*, *force_default=False*)

Bases: *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.GetObjectClassesMixin*, *AddItemsDialog*

A dialog to query user's preferences for new objects.

Parameters

- **parent** (SpineDBEditor) –
- **parent_item** (MultiDBTreeItem) –
- **db_mngr** (SpineDBManager) –
- ***db_maps** – DiffDatabaseMapping instances
- **force_default** (*bool*) – if True, defaults are non-editable

accept()

Collect info from dialog and try to add items.

class spinetoolbox.spine_db_editor.widgets.add_items_dialogs.**AddRelationshipClassesDialog**(*parent*, *parent_item*, *db_mngr*, **db_maps*, *force_default=Fa*

Bases: *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.ShowIconColorEditorMixin*, *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.GetObjectClassesMixin*, *AddItemsDialog*

A dialog to query user's preferences for new relationship classes.

Parameters

- **parent** (SpineDBEditor) –
- **parent_item** (MultiDBTreeItem) –
- **db_mngr** (SpineDBManager) –
- ***db_maps** – DiffDatabaseMapping instances
- **force_default** (*bool*) – if True, defaults are non-editable

connect_signals()

Connect signals to slots.

_handle_spin_box_value_changed(*i*)

insert_column()

remove_column()

_handle_model_data_changed(*top_left*, *bottom_right*, *roles*)

Reimplement in subclasses to handle changes in model data.

accept()

Collect info from dialog and try to add items.

class spinetoolbox.spine_db_editor.widgets.add_items_dialogs.**AddOrManageRelationshipsDialog**(*parent*, *db_mngr*, **db_maps*)

Bases: *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.GetRelationshipClassesMixin*, *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.GetObjectsMixin*, *AddItemsDialog*

A dialog to query user's preferences for new relationships.

Parameters

- **parent** (SpineDBEditor) –
- **db_mngr** (SpineDBManager) –
- ***db_maps** – DiffDatabaseMapping instances

abstract make_model()

connect_signals()

Connect signals to slots.

abstract reset_model(*index*)

Called when relationship_class's combobox's index changes. Update relationship_class attribute accordingly and reset model.

class spinetoolbox.spine_db_editor.widgets.add_items_dialogs.**AddRelationshipsDialog**(*parent*, *parent_item*, *db_mngr*, **db_maps*, *force_default=False*)

Bases: *AddOrManageRelationshipsDialog*

A dialog to query user's preferences for new relationships.

Parameters

- **parent** (SpineDBEditor) –
- **parent_item** (MultiDBTreeItem) –
- **db_mngr** (SpineDBManager) –
- ***db_maps** – DiffDatabaseMapping instances
- **force_default** (*bool*) – if True, defaults are non-editable

make_model()

reset_model(*index*)

Setup model according to current relationship_class selected in combobox.

_handle_model_data_changed(*top_left*, *bottom_right*, *roles*)

Reimplement in subclasses to handle changes in model data.

accept()

Collect info from dialog and try to add items.

class spinetoolbox.spine_db_editor.widgets.add_items_dialogs.**ManageRelationshipsDialog**(*parent*, *parent_item*, *db_mngr*, **db_maps*)

Bases: *AddOrManageRelationshipsDialog*

A dialog to query user's preferences for managing relationships.

Parameters

- **parent** (SpineDBEditor) – data store widget
- **parent_item** (MultiDBTreeItem) –
- **db_mngr** (SpineDBManager) – the manager to do the removal
- ***db_maps** – DiffDatabaseMapping instances
- **relationship_class_key** (*str, optional*) – relationships class name, object_class name list string.

make_model()

splitter_widgets()

connect_signals()

Connect signals to slots.

reset_relationship_class_combo_box(*database*, *relationship_class_key=None*)

add_relationships(*checked=True*)

reset_model(*index*)

Setup model according to current relationship_class selected in combobox.

resize_window_to_columns(*height=None*)

accept()

Collect info from dialog and try to add items.

class spinetoolbox.spine_db_editor.widgets.add_items_dialogs.**ObjectGroupDialogBase**(*parent*, *object_class_item*, *db_mngr*, **db_maps*)

Bases: PySide2.QtWidgets.QDialog

Parameters

- **parent** (SpineDBEditor) – data store widget
- **object_class_item** (ObjectClassItem) –
- **db_mngr** (SpineDBManager) –
- ***db_maps** – database mappings

connect_signals()

Connect signals to slots.

reset_list_widgets(*database*)

abstract initial_member_ids()

abstract initial_entity_id()

add_members(*checked=False*)

remove_members(*checked=False*)

_check_validity()

class spinetoolbox.spine_db_editor.widgets.add_items_dialogs.**AddObjectGroupDialog**(*parent*, *object_class_item*, *db_mngr*, **db_maps*)

Bases: *ObjectGroupDialogBase*

Parameters

- **parent** (SpineDBEditor) – data store widget
- **object_class_item** (ObjectClassItem) –
- **db_mngr** (SpineDBManager) –
- ***db_maps** – database mappings

initial_member_ids()

initial_entity_id()

_check_validity()

accept()

class spinetoolbox.spine_db_editor.widgets.add_items_dialogs.**ManageMembersDialog**(*parent*, *object_item*, *db_mngr*, **db_maps*)

Bases: *ObjectGroupDialogBase*

Parameters

- **parent** (SpineDBEditor) – data store widget
- **object_item** (entity_tree_item.ObjectItem) –
- **db_mngr** (SpineDBManager) –
- ***db_maps** – database mappings

_entity_groups()

initial_member_ids()

initial_entity_id()

accept()

spinetoolbox.spine_db_editor.widgets.commit_viewer

Contains the CommitViewer class.

author

M. Marin (KTH)

date

26.11.2018

Module Contents

Classes

_DBCommitViewer	
_CommitItem	A widget to show commit message, author and data on a QTreeWidget.
_AffectedItemsFromOneTable	A widget to show all the items from one table that are affected by a commit.
CommitViewer	**param qsettings**

class spinetoolbox.spine_db_editor.widgets.commit_viewer.**_DBCommitViewer**(*db_mngr*, *db_map*, *parent=None*)

Bases: PySide2.QtWidgets.QWidget

_select_commit(*current*, *previous*)

_do_select_commit(*current*)

class spinetoolbox.spine_db_editor.widgets.commit_viewer.**_CommitItem**(*commit*, *parent=None*)

Bases: PySide2.QtWidgets.QWidget

A widget to show commit message, author and data on a QTreeWidget.

class spinetoolbox.spine_db_editor.widgets.commit_viewer.**_AffectedItemsFromOneTable**(*items*, *parent=None*)

Bases: PySide2.QtWidgets.QTreeWidget

A widget to show all the items from one table that are affected by a commit.

moveEvent(*ev*)

sizeHint()

class spinetoolbox.spine_db_editor.widgets.commit_viewer.**CommitViewer**(*qsettings*, *db_mngr*, **db_maps*, *parent=None*)

Bases: PySide2.QtWidgets.QMainWindow

Parameters

- **qsettings** (*QSettings*) –
- **db_mngr** (SpineDBManager) –
- **db_maps** (*DiffDatabaseMapping*) –

_carry_splitter_state(*index*)

closeEvent(*ev*)

spinetoolbox.spine_db_editor.widgets.custom_delegates

Custom item delegates.

author

M. Marin (KTH)

date

1.9.2018

Module Contents

Classes

RelationshipPivotTableDelegate	A delegate that places a fully functioning QCheckBox.
ScenarioAlternativeTableDelegate	A delegate that places a QCheckBox but draws a number instead of the check.
ParameterPivotTableDelegate	**param parent**
ParameterValueElementDelegate	Delegate for Array and Map editors' table cells.
ParameterDelegate	Base class for all custom parameter delegates.
DatabaseNameDelegate	A delegate for the database name.
ParameterValueOrDefaultValueDelegate	A delegate for either the value or the default value.
ParameterDefaultValueDelegate	A delegate for the default value.
ParameterValueDelegate	A delegate for the parameter_value.
ValueListDelegate	A delegate for the parameter value list.
ObjectClassNameDelegate	A delegate for the object_class name.
RelationshipClassNameDelegate	A delegate for the relationship_class name.
ParameterNameDelegate	A delegate for the object parameter name.
ObjectNameDelegate	A delegate for the object name.
AlternativeNameDelegate	A delegate for the object name.
ObjectNameListDelegate	A delegate for the object name list.
ToolFeatureDelegate	A delegate for the tool feature tree.
AlternativeScenarioDelegate	A delegate for the alternative scenario tree.
ParameterValueListDelegate	A delegate for the parameter value list tree.
ManageItemsDelegate	A custom delegate for the model in {Add/Edit}ItemDialogs.
ManageEntityClassesDelegate	A custom delegate for the model in {Add/Edit}ItemDialogs.
ManageObjectClassesDelegate	A delegate for the model and view in {Add/Edit}ObjectClassesDialog.
ManageObjectsDelegate	A delegate for the model and view in {Add/Edit}ObjectsDialog.
ManageRelationshipClassesDelegate	A delegate for the model and view in {Add/Edit}RelationshipClassesDialog.
ManageRelationshipsDelegate	A delegate for the model and view in {Add/Edit}RelationshipsDialog.
RemoveEntitiesDelegate	A delegate for the model and view in RemoveEntitiesDialog.
ItemMetadataDelegate	A delegate for name and value columns in item metadata editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**RelationshipPivotTableDelegate**(*parent*)

Bases: *spinetoolbox.widgets.custom_delegates.CheckBoxDelegate*

A delegate that places a fully functioning QCheckBox.

Parameters
parent (SpineDBEditor) –

data_committed

static _is_relationship_index(*index*)

Checks whether or not the given index corresponds to a relationship, in which case we need to use the check

box delegate.

Returns
bool

setModelData(*editor*, *model*, *index*)

Send signal.

setEditorData(*editor*, *index*)

Do nothing. We're setting editor data right away in createEditor.

paint(*painter*, *option*, *index*)

Paint a checkbox without the label.

editorEvent(*event*, *model*, *option*, *index*)

Change the data in the model and the state of the checkbox when user presses left mouse button and this cell is editable. Otherwise do nothing.

createEditor(*parent*, *option*, *index*)

Important, otherwise an editor is created if the user clicks in this cell. ** Need to hook up a signal to the model.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ScenarioAlternativeTableDelegate**(*parent*)

Bases: *spinetoolbox.widgets.custom_delegates.RankDelegate*

A delegate that places a QCheckBox but draws a number instead of the check.

Parameters
parent (SpineDBEditor) –

data_committed

static _is_scenario_alternative_index(*index*)

Checks whether or not the given index corresponds to a scenario alternative, in which case we need to use the rank delegate.

Returns
bool

setModelData(*editor*, *model*, *index*)

Send signal.

setEditorData(*editor*, *index*)

Do nothing. We're setting editor data right away in createEditor.

paint(*painter*, *option*, *index*)

Paint a checkbox without the label.

editorEvent(*event*, *model*, *option*, *index*)

Change the data in the model and the state of the checkbox when user presses left mouse button and this cell is editable. Otherwise do nothing.

createEditor(*parent*, *option*, *index*)

Important, otherwise an editor is created if the user clicks in this cell. ** Need to hook up a signal to the model.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ParameterPivotTableDelegate**(*parent*)

Bases: PySide2.QtWidgets.QStyledItemDelegate

Parameters
parent (SpineDBEditor) –

parameter_value_editor_requested

data_committed

setModelData(*editor*, *model*, *index*)

Send signal.

setEditorData(*editor*, *index*)

Do nothing. We're setting editor data right away in createEditor.

createEditor(*parent*, *option*, *index*)

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ParameterValueElementDelegate**

Bases: PySide2.QtWidgets.QStyledItemDelegate

Delegate for Array and Map editors' table cells.

value_editor_requested

Emitted when editing the value requires the full blown editor dialog.

setModelData(*editor*, *model*, *index*)

Sets data in the model.

editor (CustomLineEditor): editor widget model (QAbstractItemModel): model index (QModelIndex): target index

createEditor(*parent*, *option*, *index*)

Creates an editor widget or emits value_editor_requested for complex values.

Parameters

- **parent** (*QWidget*) – parent widget
- **option** (*QStyleOptionViewItem*) – unused
- **index** (*QModelIndex*) – element's model index

Returns
editor widget

Return type
ParameterValueLineEditor

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ParameterDelegate**(*parent*, *db_mngr*)

Bases: PySide2.QtWidgets.QStyledItemDelegate

Base class for all custom parameter delegates.

db_mngr

database manager

Type
SpineDBManager

Parameters

- **parent** (*QWidget*) – parent widget
- **db_mngr** (SpineDBManager) – database manager

data_committed

setModelData(*editor*, *model*, *index*)

Send signal.

setEditorData(*editor*, *index*)

Do nothing. We're setting editor data right away in createEditor.

updateEditorGeometry(*editor*, *option*, *index*)

_close_editor(*editor*, *index*)

Closes editor. Needed by SearchBarEditor.

_get_db_map(*index*)

Returns the db_map for the database at given index or None if not set yet.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**DatabaseNameDelegate**(*parent*, *db_mngr*)

Bases: *ParameterDelegate*

A delegate for the database name.

Parameters

- **parent** (*QWidget*) – parent widget
- **db_mngr** (SpineDBManager) – database manager

createEditor(*parent*, *option*, *index*)

Returns editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ParameterValueOrDefaultValueDelegate**(*parent*, *db_mng*

Bases: *ParameterDelegate*

A delegate for either the value or the default value.

Parameters

- **parent** (*QWidget*) – parent widget
- **db_mngr** (SpineDBManager) – database manager

parameter_value_editor_requested

setModelData(*editor*, *model*, *index*)

Send signal.

_create_or_request_parameter_value_editor(*parent*, *option*, *index*, *db_map*)

Emits the signal to request a standalone *ParameterValueEditor* from parent widget.

createEditor(*parent*, *option*, *index*)

If the parameter has associated a value list, returns a SearchBarEditor. Otherwise returns or requests a dedicated parameter_value editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ParameterDefaultValueDelegate**(*parent*, *db_mngr*)

Bases: *ParameterValueOrDefaultValueDelegate*

A delegate for the default value.

Parameters

- **parent** (*QWidget*) – parent widget
- **db_mngr** (SpineDBManager) – database manager

_get_value_list_id(*index*, *db_map*)

Returns a value list item for the given index and db_map.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ParameterValueDelegate**(*parent*, *db_mngr*)

Bases: *ParameterValueOrDefaultValueDelegate*

A delegate for the parameter_value.

Parameters

- **parent** (*QWidget*) – parent widget
- **db_mngr** (SpineDBManager) – database manager

_get_value_list_id(*index*, *db_map*)

Returns a value list item for the given index and db_map.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ValueListDelegate**(*parent*, *db_mngr*)

Bases: *ParameterDelegate*

A delegate for the parameter value list.

Parameters

- **parent** (*QWidget*) – parent widget
- **db_mngr** (SpineDBManager) – database manager

createEditor(*parent*, *option*, *index*)

Returns editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ObjectClassNameDelegate**(*parent*, *db_mngr*)

Bases: *ParameterDelegate*

A delegate for the object_class name.

Parameters

- **parent** (*QWidget*) – parent widget
- **db_mngr** (SpineDBManager) – database manager

createEditor(*parent*, *option*, *index*)

Returns editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**RelationshipClassNameDelegate**(*parent*, *db_mngr*)

Bases: *ParameterDelegate*

A delegate for the relationship_class name.

Parameters

- **parent** (*QWidget*) – parent widget
- **db_mngr** (SpineDBManager) – database manager

createEditor(*parent*, *option*, *index*)

Returns editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ParameterNameDelegate**(*parent*, *db_mngr*)

Bases: *ParameterDelegate*

A delegate for the object parameter name.

Parameters

- **parent** (*QWidget*) – parent widget
- **db_mngr** (SpineDBManager) – database manager

createEditor(*parent*, *option*, *index*)

Returns editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ObjectNameDelegate**(*parent*, *db_mngr*)

Bases: *ParameterDelegate*

A delegate for the object name.

Parameters

- **parent** (*QWidget*) – parent widget
- **db_mngr** (SpineDBManager) – database manager

createEditor(*parent*, *option*, *index*)

Returns editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**AlternativeNameDelegate**(*parent*, *db_mngr*)

Bases: *ParameterDelegate*

A delegate for the object name.

Parameters

- **parent** (*QWidget*) – parent widget
- **db_mngr** (SpineDBManager) – database manager

createEditor(*parent*, *option*, *index*)

Returns editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ObjectNameListDelegate**(*parent*, *db_mngr*)

Bases: *ParameterDelegate*

A delegate for the object name list.

Parameters

- **parent** (*QWidget*) – parent widget
- **db_mngr** (SpineDBManager) – database manager

object_name_list_editor_requested

createEditor(*parent*, *option*, *index*)

Returns editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ToolFeatureDelegate**(**args*, ***kwargs*)

Bases: PySide2.QtWidgets.QStyledItemDelegate

A delegate for the tool feature tree.

data_committed

_get_names(*item*, *model*)

static _get_index_data(*item*, *index*)

setModelData(*editor*, *model*, *index*)

Send signal.

setEditorData(*editor*, *index*)

Do nothing. We're setting editor data right away in createEditor.

createEditor(*parent*, *option*, *index*)

Returns editor.

updateEditorGeometry(*editor*, *option*, *index*)

_close_editor(*editor*, *index*)

Closes editor. Needed by SearchBarEditor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**AlternativeScenarioDelegate**(**args*, ***kwargs*)

Bases: PySide2.QtWidgets.QStyledItemDelegate

A delegate for the alternative scenario tree.

data_committed

setModelData(*editor*, *model*, *index*)

Send signal.

setEditorData(*editor*, *index*)

Do nothing. We're setting editor data right away in createEditor.

_get_names(*item*, *model*)

static _get_index_data(*item*, *index*)

createEditor(*parent*, *option*, *index*)

Returns editor.

updateEditorGeometry(*editor*, *option*, *index*)

_close_editor(*editor*, *index*)

Closes editor. Needed by SearchBarEditor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ParameterValueListDelegate**

Bases: PySide2.QtWidgets.QStyledItemDelegate

A delegate for the parameter value list tree.

data_committed

parameter_value_editor_requested

setModelData(*editor*, *model*, *index*)

Send signal.

setEditorData(*editor*, *index*)

Do nothing. We're setting editor data right away in createEditor.

createEditor(*parent*, *option*, *index*)

Returns editor.

_close_editor(*editor*, *index*)

Closes editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ManageItemsDelegate**

Bases: PySide2.QtWidgets.QStyledItemDelegate

A custom delegate for the model in {Add/Edit}ItemDialogs.

data_committed

setModelData(*editor*, *model*, *index*)

Send signal.

close_editor(*editor*, *index*, *model*)

updateEditorGeometry(*editor*, *option*, *index*)

connect_editor_signals(*editor*, *index*)

Connect editor signals if necessary.

_create_database_editor(*parent*, *option*, *index*)

createEditor(*parent*, *option*, *index*)

Returns an editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ManageEntityClassesDelegate**

Bases: *ManageItemsDelegate*

A custom delegate for the model in {Add/Edit}ItemDialogs.

paint(*painter*, *option*, *index*)

Get a pixmap from the index data and paint it in the middle of the cell.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ManageObjectClassesDelegate**

Bases: *ManageEntityClassesDelegate*

A delegate for the model and view in {Add/Edit}ObjectClassesDialog.

icon_color_editor_requested

createEditor(*parent*, *option*, *index*)

Return editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ManageObjectsDelegate**

Bases: *ManageItemsDelegate*

A delegate for the model and view in {Add/Edit}ObjectsDialog.

createEditor(*parent*, *option*, *index*)

Return editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ManageRelationshipClassesDelegate**

Bases: *ManageEntityClassesDelegate*

A delegate for the model and view in {Add/Edit}RelationshipClassesDialog.

icon_color_editor_requested

createEditor(*parent*, *option*, *index*)

Return editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ManageRelationshipsDelegate**

Bases: *ManageItemsDelegate*

A delegate for the model and view in {Add/Edit}RelationshipsDialog.

createEditor(*parent*, *option*, *index*)

Return editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**RemoveEntitiesDelegate**

Bases: *ManageItemsDelegate*

A delegate for the model and view in RemoveEntitiesDialog.

createEditor(*parent*, *option*, *index*)

Return editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ItemMetadataDelegate**(*item_metadata_model*, *metadata_model*, *column*, *parent*)

Bases: PySide2.QtWidgets.QStyledItemDelegate

A delegate for name and value columns in item metadata editor.

Parameters

- **item_metadata_model** (*ItemMetadataModel*) – item metadata model
- **metadata_model** (MetadataTableModel) – metadata model
- **column** (*int*) – item metadata table column column
- **parent** (*QObject, optional*) – parent object

createEditor(*parent*, *option*, *index*)

spinetoolbox.spine_db_editor.widgets.custom_menus

Classes for custom context menus and pop-up menus.

author

M. Marin (KTH)

date

13.5.2020

Module Contents

Classes

MainMenu	
ParameterViewFilterMenu	Filter menu.
TabularViewFilterMenu	Filter menu to use together with FilterWidget in TabularViewMixin.

class spinetoolbox.spine_db_editor.widgets.custom_menus.**MainMenu**

Bases: PySide2.QtWidgets.QMenu

event(*ev*)

Intercepts shortcuts and instead sends an equivalent event with the 'Alt' modifier, so that mnemonics works with just the key. Also sends a key press event with the 'Alt' key when this menu shows, so that mnemonics are underlined on Windows.

class spinetoolbox.spine_db_editor.widgets.custom_menus.**ParameterViewFilterMenu**(*parent*, *source_model*, *field*, *show_empty=True*)

Bases: *spinetoolbox.widgets.custom_menus.FilterMenuBase*

Filter menu.

Parameters

- **parent** (SpineDBEditor) –
- **source_model** (CompoundParameterModel) – a model to lazily get data from
- **field** (*str*) – the field name

filterChanged

_handle_source_model_refreshed()

Updates the menu to only present values that are actually shown in the source model.

set_filter_accepted_values(*accepted_values*)

set_filter_rejected_values(*rejected_values*)

_get_value_to_remove(*action*, *db_map*, *db_item*)

_get_value_to_add(*action*, *db_map*, *db_item*)

modify_menu_data(*action*, *db_map*, *db_items*)

Modifies data in the menu.

Parameters

- **action** (*str*) – either 'add', 'remove', or 'update'
- **db_map** (*DiffDatabaseMapping*) –
- **db_items** (*list(dict)*) –

_build_auto_filter(*valid_values*)

Builds the auto filter given valid values.

Parameters
valid_values (*Sequence*) – Values accepted by the filter.

Returns
mapping db_map, to entity_class_id, to set of accepted parameter_value/definition ids

Return type
dict

emit_filter_changed(*valid_values*)

Builds auto filter and emits signal.

Parameters
valid_values (*Sequence*) – Values accepted by the filter.

class spinetoolbox.spine_db_editor.widgets.custom_menus.**TabularViewFilterMenu**(*parent*, *identifier*, *data_to_value*, *show_empty=True*)

Bases: *spinetoolbox.widgets.custom_menus.FilterMenuBase*

Filter menu to use together with FilterWidget in TabularViewMixin.

Parameters

- **parent** (SpineDBEditor) –
- **identifier** (*int*) – index identifier
- **data_to_value** (*method*) – a method to translate item data to a value for display role

filterChanged

emit_filter_changed(*valid_values*)

event(*event*)

spinetoolbox.spine_db_editor.widgets.custom_qgraphicsviews

Classes for custom QGraphicsViews for the Entity graph view.

authors

P. Savolainen (VTT), M. Marin (KTH)

date

6.2.2018

Module Contents

Classes

EntityQGraphicsView	QGraphicsView for the Entity Graph View.

class spinetoolbox.spine_db_editor.widgets.custom_qgraphicsviews.**EntityQGraphicsView**(*parent*)

Bases: *spinetoolbox.widgets.custom_qgraphicsviews.CustomQGraphicsView*

QGraphicsView for the Entity Graph View.

Parameters

parent (*QWidget*) – Graph View Form's (QMainWindow) central widget (self.centralwidget)

property _qsettings

property db_mngr

property entity_items

graph_selection_changed

handle_scene_selection_changed()

Filters parameters by selected objects in the graph.

connect_spine_db_editor(*spine_db_editor*)

populate_context_menu()

increase_arc_length()

decrease_arc_length()

_update_actions_visibility()

Enables or disables actions according to current selection in the graph.

make_items_menu()

_set_auto_expand_objects(*_checked=False*, *save_setting=True*)

set_auto_expand_objects(*checked*)

_set_merge_dbs(*_checked=False*, *save_setting=True*)

set_merge_dbs(*checked*)

_set_disable_max_relationship_dimension(*_checked=False*, *save_setting=True*)

set_disable_max_relationship_dimension(*checked*)

_set_max_relationship_dimension(*_value=None*, *save_setting=True*)

set_max_relationship_dimension(*value*)

add_objects_at_position(*checked=False*)

edit_selected(*_=False*)

Edits selected items.

remove_selected(*_=False*)

Removes selected items.

_get_selected_entity_names()

hide_selected_items(*checked=False*)

Hides selected items.

_hide_class(*action*)

Hides some class.

show_all_hidden_items(*checked=False*)

Shows all hidden items.

show_hidden_items(*action*)

Shows some hidden items.

prune_selected_items(*checked=False*)

Prunes selected items.

_prune_class(*action*)

Prunnes some class.

restore_all_pruned_items(*checked=False*)

Reinstates all pruned items.

restore_pruned_items(*action*)

Reinstates some pruned items.

select_position_parameters(*checked=False*)

_set_position_parameters(*parameter_pos_x*, *parameter_pos_y*)

save_positions(*checked=False*)

clear_saved_positions(*checked=False*)

export_as_pdf(*checked=False*)

_populate_add_heat_map_menu()

Populates the menu 'Add heat map' with parameters for currently shown items in the graph.

add_heat_map(*action*)

Adds heat map for the parameter in the action text.

_clean_up_heat_map_items()

set_cross_hairs_items(*relationship_class*, *cross_hairs_items*)

Sets 'cross_hairs' items for relationship creation.

Parameters

- **relationship_class** (*dict*) –
- **cross_hairs_items** (*list(QGraphicsItems)*) –

clear_cross_hairs_items()

_cross_hairs_has_valid_target()

mousePressEvent(*event*)

Handles relationship creation if one it's in process.

mouseMoveEvent(*event*)

Updates the hovered object item if we're in relationship creation mode.

_update_cross_hairs_pos(*pos*)

Updates the hovered object item and sets the 'cross_hairs' icon accordingly.

Parameters

pos (*QPoint*) – the desired position in view coordinates

mouseReleaseEvent(*event*)

Reestablish scroll hand drag mode.

_scroll_scene_by(*dx*, *dy*)

keyPressEvent(*event*)

Aborts relationship creation if user presses ESC.

contextMenuEvent(*e*)

Shows context menu.

Parameters

e (*QContextMenuEvent*) – Context menu event

_compute_max_zoom()

_use_smooth_zoom()

_zoom(*factor*)

apply_zoom()

wheelEvent(*event*)

Zooms in/out. If user has pressed the shift key, rotates instead.

Parameters

event (*QWheelEvent*) – Mouse wheel event

_handle_rotation_time_line_advanced(*pos*)

Performs rotation whenever the smooth rotation time line advances.

_rotate(*angle*)

rotate_clockwise()

Performs a rotate clockwise with fixed angle.

rotate_anticlockwise()

Performs a rotate anticlockwise with fixed angle.

spinetoolbox.spine_db_editor.widgets.custom_qtableview

Custom QTableView classes that support copy-paste and the like.

author

M. Marin (KTH)

date

18.5.2018

Module Contents

Classes

ParameterTableView	Custom QTableView class with autofilter functionality.
ObjectParameterTableMixin	
RelationshipParameterTableMixin	
ParameterDefinitionTableView	Custom QTableView class with autofilter functionality.
ParameterValueTableView	Custom QTableView class with autofilter functionality.
ObjectParameterDefinitionTableView	A custom QTableView for the object parameter_definition pane in Spine db editor.
RelationshipParameterDefinitionTableView	A custom QTableView for the relationship parameter_definition pane in Spine db editor.
ObjectParameterValueTableView	A custom QTableView for the object parameter_value pane in Spine db editor.
RelationshipParameterValueTableView	A custom QTableView for the relationship parameter_value pane in Spine db editor.
PivotTableView	Custom QTableView class with pivot capabilities.
FrozenTableView	
MetadataTableViewBase	Base for metadata and item metadata table views.
MetadataTableView	Table view for metadata.
ItemMetadataTableView	Table view for entity and parameter value metadata.

Functions

_set_parameter_data(index, new_value)	Updates (object or relationship) parameter_definition or value with newly edited data.

spinetoolbox.spine_db_editor.widgets.custom_qtableview.**_set_parameter_data**(*index*, *new_value*)

Updates (object or relationship) parameter_definition or value with newly edited data.

class spinetoolbox.spine_db_editor.widgets.custom_qtableview.**ParameterTableView**(*parent*)

Bases: *spinetoolbox.widgets.custom_qtableview.AutoFilterCopyPasteTableView*

Custom QTableView class with autofilter functionality.

Parameters

parent (*QObject*) – parent object

value_column_header :str

Either "default value" or "value". Used to identify the value column for advanced editing and plotting.

connect_spine_db_editor(*spine_db_editor*)

Connects a Spine db editor to work with this view.

Parameters

spine_db_editor (SpineDBEditor) –

_make_delegate(*column_name*, *delegate_class*)

Creates a delegate for the given column and returns it.

Parameters

- **column_name** (*str*) –
- **delegate_class** (ParameterDelegate) –

Returns

ParameterDelegate

create_delegates()

Creates delegates for this view

open_in_editor()

Opens the current index in a parameter_value editor using the connected Spine db editor.

plot(*checked=False*)

Plots current index.

abstract _plot_selection(*selection*, *plot_widget=None*)

Adds selected indexes to existing plot or creates a new plot window.

Parameters

- **selection** (*Iterable of QModelIndex*) – a list of QModelIndex objects for plotting
- **plot_widget** (PlotWidget, *optional*) – an existing plot widget to draw into or None to create a new widget

Returns

a PlotWidget object

Return type

PlotWidget

plot_in_window(*action*)

Plots current index in the window given by action's name.

populate_context_menu()

Creates a context menu for this view.

contextMenuEvent(*event*)

Shows context menu.

Parameters

event (*QContextMenuEvent*) –

_selected_rows_per_column()

Computes selected rows per column.

Returns

Mapping columns to selected rows in that column.

Return type
dict

filter_by_selection(*checked=False*)

filter_excluding_selection(*checked=False*)

remove_selected()

Removes selected indexes.

rowsInserted(*parent*, *start*, *end*)

_refresh_copy_paste_actions(_, __)

Enables or disables copy and paste actions.

class spinetoolbox.spine_db_editor.widgets.custom_qtableview.**ObjectParameterTableMixin**

create_delegates()

class spinetoolbox.spine_db_editor.widgets.custom_qtableview.**RelationshipParameterTableMixin**

create_delegates()

class spinetoolbox.spine_db_editor.widgets.custom_qtableview.**ParameterDefinitionTableView**(*parent*)

Bases: *ParameterTableView*

Custom QTableView class with autofilter functionality.

Parameters
parent (*QObject*) – parent object

value_column_header = default_value

create_delegates()

Creates delegates for this view

abstract _plot_selection(*selection*, *plot_widget=None*)

See base class

class spinetoolbox.spine_db_editor.widgets.custom_qtableview.**ParameterValueTableView**(*parent*)

Bases: *ParameterTableView*

Custom QTableView class with autofilter functionality.

Parameters
parent (*QObject*) – parent object

abstract property _pk_fields

value_column_header = value

connect_spine_db_editor(*spine_db_editor*)

Connects a Spine db editor to work with this view.

Parameters
spine_db_editor (SpineDBEditor) –

create_delegates()

Creates delegates for this view

_update_pinned_values(*_selected*, *_deselected*)

_make_pinned_value(*index*)

abstract _plot_selection(*selection*, *plot_widget=None*)

See base class

class spinetoolbox.spine_db_editor.widgets.custom_qtableview.**ObjectParameterDefinitionTableView**(*parent*)

Bases: *ObjectParameterTableMixin*, *ParameterDefinitionTableView*

A custom QTableView for the object parameter_definition pane in Spine db editor.

Parameters
parent (*QObject*) – parent object

_plot_selection(*selection*, *plot_widget=None*)

See base class

class spinetoolbox.spine_db_editor.widgets.custom_qtableview.**RelationshipParameterDefinitionTableView**(*pa*

Bases: *RelationshipParameterTableMixin*, *ParameterDefinitionTableView*

A custom QTableView for the relationship parameter_definition pane in Spine db editor.

Parameters
parent (*QObject*) – parent object

_plot_selection(*selection*, *plot_widget=None*)

See base class

class spinetoolbox.spine_db_editor.widgets.custom_qtableview.**ObjectParameterValueTableView**(*parent*)

Bases: *ObjectParameterTableMixin*, *ParameterValueTableView*

A custom QTableView for the object parameter_value pane in Spine db editor.

Parameters
parent (*QObject*) – parent object

property _pk_fields

create_delegates()

Creates delegates for this view

_plot_selection(*selection*, *plot_widget=None*)

See base class.

class spinetoolbox.spine_db_editor.widgets.custom_qtableview.**RelationshipParameterValueTableView**(*parent*)

Bases: *RelationshipParameterTableMixin*, *ParameterValueTableView*

A custom QTableView for the relationship parameter_value pane in Spine db editor.

Parameters
parent (*QObject*) – parent object

property _pk_fields

create_delegates()

Creates delegates for this view

_plot_selection(*selection*, *plot_widget=None*)

See base class.

class spinetoolbox.spine_db_editor.widgets.custom_qtableview.**PivotTableView**(*parent=None*)

Bases: *spinetoolbox.widgets.custom_qtableview.CopyPasteTableView*

Custom QTableView class with pivot capabilities.

Uses 'contexts' to provide different UI elements (table headers, context menus,...) depending on what data the pivot table currently contains.

Parameters
parent (*QWidget, optional*) – parent widget

class _ContextBase(*view*, *db_editor*, *horizontal_header*, *vertical_header*)

Base class for pivot table view's contexts.

Parameters

- **view** (PivotTableView) – parent view
- **db_editor** (SpineDBEditor) – database editor
- **horizontal_header** (*QHeaderView*) – horizontal header
- **vertical_header** (*QHeaderView*) – vertical header

_REMOVE_OBJECT = Remove objects

_REMOVE_RELATIONSHIP = Remove relationships

_REMOVE_PARAMETER = Remove parameter definitions

_REMOVE_ALTERNATIVE = Remove alternatives

_REMOVE_SCENARIO = Remove scenarios

_DUPLICATE_SCENARIO = Duplicate scenario

_clear_selection_lists()

Clears cached selected index lists.

abstract populate_context_menu()

Generates context menu.

_refresh_selected_indexes()

Caches selected index lists.

remove_alternatives()

Removes selected alternatives from the database.

show_context_menu(*position*)

Shows the context menu.

_to_selection_lists(*index*)

Caches given index to corresponding selected index list.
Parameters
index (*QModelIndex*) – index to cache

abstract _update_actions_availability()

Enables/disables context menu entries before the menu is shown.

class _EntityContextBase(*view*, *db_editor*, *horizontal_header*, *vertical_header*)

Bases: *PivotTableView._ContextBase*

Base class for contexts that contain entities and entity classes.

Parameters

- **view** (PivotTableView) – parent view
- **db_editor** (SpineDBEditor) – database editor
- **horizontal_header** (*QHeaderView*) – horizontal header
- **vertical_header** (*QHeaderView*) – vertical header

_can_remove_relationships()

Checks if it makes sense to remove selected relationships from the database.

Returns

True if relationships can be removed, False otherwise

Return type

bool

_clear_selection_lists()

See base class.

abstract populate_context_menu()

See base class.

remove_objects()

Removes selected objects from the database.

remove_relationships()

Removes selected relationships from the database.

abstract _update_actions_availability()

See base class.

class _ParameterValueContext(*view*, *db_editor*)

Bases: *PivotTableView._EntityContextBase*

Context for showing parameter values in the pivot table.

Parameters

- **view** (PivotTableView) – parent view
- **db_editor** (SpineDBEditor) – database editor

_clear_selection_lists()

See base class.

populate_context_menu()

See base class.

open_in_editor()

Opens the parameter value editor for the first selected cell.

plot()

Plots the selected cells.

_plot_in_window(*action*)

Plots the selected cells in an existing window.

remove_parameters()

Removes selected parameter definitions from the database.

remove_values()

Removes selected parameter values from the database.

show_context_menu(*position*)

Shows the context menu.

_to_selection_lists(*index*)

See base class.

_update_actions_availability()

See base class.

class _IndexExpansionContext(*view*, *db_editor*)

Bases: *PivotTableView._ParameterValueContext*

Context for expanded parameter values

Parameters

- **view** (PivotTableView) – parent view
- **db_editor** (SpineDBEditor) – database editor

class _RelationshipContext(*view*, *db_editor*)

Bases: *PivotTableView._EntityContextBase*

Context for presenting relationships in the pivot table.

Parameters

- **view** (PivotTableView) – parent view
- **db_editor** (SpineDBEditor) – database editor

populate_context_menu()

See base class.

_update_actions_availability()

See base class.

class _ScenarioAlternativeContext(*view*, *db_editor*)

Bases: *PivotTableView._ContextBase*

Context for presenting scenarios and alternatives

Parameters

- **view** (PivotTableView) – parent view
- **db_editor** (SpineDBEditor) – database editor

_clear_selection_lists()

See base class.

populate_context_menu()

See base class.

remove_scenarios()

Removes selected scenarios from the database.

duplicate_scenario()

Duplicates current scenario in the database.

_to_selection_lists(*index*)

See base class.

_update_actions_availability()

See base class.

_open_scenario_generator()

Opens the scenario generator dialog.

_toggle_checked_state()

Toggles the checked state of selected alternatives.

property source_model

property db_mngr

header_changed

connect_spine_db_editor(*spine_db_editor*)

_change_context()

Changes the UI engine according to pivot model type.

contextMenuEvent(*event*)

Shows context menu.

Parameters

event (*QContextMenuEvent*) –

setModel(*model*)

setIndexWidget(*proxy_index*, *widget*)

setHorizontalHeader(*horizontal_header*)

setVerticalHeader(*vertical_header*)

resizeEvent(*ev*)

_fetch_more_visible()

_init_header_tables()

_update_header_tables()

_update_section_width(*logical_index*, *_old_size*, *new_size*)

_update_section_height(*logical_index*, *_old_size*, *new_size*)

_update_header_tables_geometry()

_refresh_copy_paste_actions(_, __)

class spinetoolbox.spine_db_editor.widgets.custom_qtableview.**FrozenTableView**

Bases: PySide2.QtWidgets.QTableView

property area

header_dropped

dragEnterEvent(*event*)

dragMoveEvent(*event*)

dropEvent(*event*)

class spinetoolbox.spine_db_editor.widgets.custom_qtableview.**MetadataTableViewBase**(*parent*)

Bases: *spinetoolbox.widgets.custom_qtableview.CopyPasteTableView*

Base for metadata and item metadata table views.

Parameters
parent (*QWidget, optional*) – parent widget

connect_spine_db_editor(*db_editor*)

Finishes view's initialization.

Parameters
db_editor (SpineDBEditor) – database editor instance

contextMenuEvent(*event*)

_remove_selected()

Removes selected rows from view's model.

_enable_delegates(*db_editor*)

Creates delegates for this view

Parameters
db_editor (SpineDBEditor) – database editor

_populate_context_menu()

Fills context menu with actions.

_set_model_data(*index*, *value*)

Sets model data.

Parameters

- **index** (*QModelIndex*) – model index to set
- **value** (*str*) – value

_refresh_copy_paste_actions()

class spinetoolbox.spine_db_editor.widgets.custom_qtableview.**MetadataTableView**(*parent*)

Bases: *MetadataTableViewBase*

Table view for metadata.

Parameters
parent (*QWidget, optional*) – parent widget

_enable_delegates(*db_editor*)

See base class.

class spinetoolbox.spine_db_editor.widgets.custom_qtableview.**ItemMetadataTableView**(*parent*)

Bases: *MetadataTableViewBase*

Table view for entity and parameter value metadata.

Parameters
parent (*QWidget*) – parent widget

set_models(*item_metadata_model*, *metadata_model*)

Sets models.

Parameters

- **item_metadata_model** (*ItemMetadataModel*) – item metadata model
- **metadata_model** (MetadataTableModel) – metadata model

_enable_delegates(*db_editor*)

See base class

spinetoolbox.spine_db_editor.widgets.custom_qtreeview

Classes for custom QTreeView.

author

M. Marin (KTH)

date
25.4.2018

Module Contents

Classes

EntityTreeView	Tree view base class for object and relationship tree views.
ObjectTreeView	Custom QTreeView class for the object tree in SpineDBEditor.
RelationshipTreeView	Custom QTreeView class for the relationship tree in SpineDBEditor.
ItemTreeView	Base class for all non-entity tree views.
ToolFeatureTreeView	Custom QTreeView class for tools and features in SpineDBEditor.
AlternativeScenarioTreeView	Custom QTreeView class for the alternative scenario tree in SpineDBEditor.
ParameterValueListTreeView	Custom QTreeView class for parameter_value_list in SpineDBEditor.

class spinetoolbox.spine_db_editor.widgets.custom_qtreeview.**EntityTreeView**(*parent*)

Bases: *spinetoolbox.widgets.custom_qtreeview.CopyTreeView*

Tree view base class for object and relationship tree views.

Initialize the view.

tree_selection_changed

reset()

connect_spine_db_editor(*spine_db_editor*)

Connects a Spine db editor to work with this view.

Parameters
spine_db_editor (SpineDBEditor) –

_add_middle_actions()

Adds action at the middle of the context menu. Subclasses can reimplement at will.

_create_context_menu()

Creates a context menu for this view.

edit(*index*, *trigger*, *event*)

Edit all selected items.

connect_signals()

Connects signals.

rowsInserted(*parent*, *start*, *end*)

rowsRemoved(*parent*, *start*, *end*)

setModel(*model*)

_fetch_more_visible()

verticalScrollbarValueChanged(*value*)

_handle_selection_changed(*selected*, *deselected*)

Classifies selection by item type and emits signal.

_refresh_selected_indexes()

clear_any_selections()

Clears the selection if any.

fully_expand()

Expands selected indexes and all their children.

fully_collapse()

Collapses selected indexes and all their children.

export_selected()

Exports data from selected indexes using the connected Spine db editor.

remove_selected()

Removes selected indexes using the connected Spine db editor.

manage_relationships()

contextMenuEvent(*event*)

Shows context menu.

Parameters
event (*QContextMenuEvent*) –

mousePressEvent(*event*)

Overrides selection behaviour if the user has selected sticky selection in Settings. If sticky selection is enabled, multiple-selection is enabled when selecting items in the Object tree. Pressing the Ctrl-button down, enables single selection.

Parameters
event (*QMouseEvent*) –

_add_relationship_actions()

update_actions_availability()

Updates the visible property of actions according to whether or not they apply to given item.

edit_selected()

Edits all selected indexes using the connected Spine db editor.

class spinetoolbox.spine_db_editor.widgets.custom_qtreeview.**ObjectTreeView**(*parent*)

Bases: *EntityTreeView*

Custom QTreeView class for the object tree in SpineDBEditor.

Initialize the view.

update_actions_availability()

Updates the visible property of actions according to whether or not they apply to given item.

_add_middle_actions()

Adds action at the middle of the context menu. Subclasses can reimplement at will.

connect_signals()

Connects signals.

rowsInserted(*parent*, *start*, *end*)

add_object_classes()

add_objects()

add_relationship_classes()

add_relationships()

find_next_relationship()

Finds the next occurrence of the relationship at the current index and expands it.

_do_find_next_relationship()

duplicate_object()

Duplicates the object at the current index using the connected Spine db editor.

add_object_group()

manage_members()

class spinetoolbox.spine_db_editor.widgets.custom_qtreeview.**RelationshipTreeView**(*parent*)

Bases: *EntityTreeView*

Custom QTreeView class for the relationship tree in SpineDBEditor.

Initialize the view.

_add_middle_actions()

Adds action at the middle of the context menu. Subclasses can reimplement at will.

update_actions_availability()

Updates the visible property of actions according to whether or not they apply to given item.

add_relationship_classes()

add_relationships()

class spinetoolbox.spine_db_editor.widgets.custom_qtreeview.**ItemTreeView**(*parent*)

Bases: *spinetoolbox.widgets.custom_qtreeview.CopyTreeView*

Base class for all non-entity tree views.

Initialize the view.

rowsInserted(*parent*, *start*, *end*)

connect_signals()

Connects signals.

abstract remove_selected()

Removes items selected in the view.

abstract update_actions_availability(*item*)

Updates the visible property of actions according to whether or not they apply to given item.

connect_spine_db_editor(*spine_db_editor*)

populate_context_menu()

Creates a context menu for this view.

contextMenuEvent(*event*)

Shows context menu.

Parameters

event (*QContextMenuEvent*) –

_refresh_copy_paste_actions(_, __)

Refreshes copy and paste actions enabled state.

class spinetoolbox.spine_db_editor.widgets.custom_qtreeview.**ToolFeatureTreeView**(*parent*)

Bases: *ItemTreeView*

Custom QTreeView class for tools and features in SpineDBEditor.

Initialize the view.

connect_spine_db_editor(*spine_db_editor*)

see base class

remove_selected()

See base class.

update_actions_availability(*item*)

See base class.

dragMoveEvent(*event*)

dragEnterEvent(*event*)

class spinetoolbox.spine_db_editor.widgets.custom_qtreeview.**AlternativeScenarioTreeView**(*parent*)

Bases: *ItemTreeView*

Custom QTreeView class for the alternative scenario tree in SpineDBEditor.

Initialize the view.

alternative_selection_changed

reset()

connect_signals()

Connects signals.

connect_spine_db_editor(*spine_db_editor*)

see base class

populate_context_menu()

See base class.

_db_map_alt_ids_from_selection(*selection*)

_db_map_scen_alt_ids_from_selection(*selection*)

_handle_selection_changed(*selected*, *deselected*)

Emits alternative_selection_changed with the current selection.

remove_selected()

See base class.

update_actions_availability(*item*)

See base class.

dragMoveEvent(*event*)

dragEnterEvent(*event*)

_open_scenario_generator()

Opens the scenario generator dialog.

class spinetoolbox.spine_db_editor.widgets.custom_qtreeview.**ParameterValueListTreeView**(*parent*)

Bases: *ItemTreeView*

Custom QTreeView class for parameter_value_list in SpineDBEditor.

Initialize the view.

connect_spine_db_editor(*spine_db_editor*)

see base class

populate_context_menu()

Creates a context menu for this view.

update_actions_availability(*item*)

See base class.

open_in_editor()

Opens the parameter_value editor for the first selected cell.

remove_selected()

See base class.

spinetoolbox.spine_db_editor.widgets.custom_qwidgets

Custom QWidgets.

author

M. Marin (KTH)

date

13.5.2018

Module Contents

Classes

DataToValueFilterWidget	Filter widget class.
LazyFilterWidget	Filter widget class.
OpenFileButton	A button to open files or show them in the folder.
OpenSQLiteFileButton	A button to open sqlite files, show them in the folder, or add them to the project.
ShootingLabel	

class spinetoolbox.spine_db_editor.widgets.custom_qwidgets.**DataToValueFilterWidget**(*parent*, *data_to_value*, *show_empty=True*)

Bases: *spinetoolbox.widgets.custom_qwidgets.FilterWidgetBase*

Filter widget class.

Init class.

Parameters

- **parent** (*QWidget*) –
- **data_to_value** (*method*) – a method to translate item data to a value for display role

class spinetoolbox.spine_db_editor.widgets.custom_qwidgets.**LazyFilterWidget**(*parent*, *source_model*, *show_empty=True*)

Bases: *spinetoolbox.widgets.custom_qwidgets.FilterWidgetBase*

Filter widget class.

Init class.

Parameters

- **parent** (SpineDBEditor) –
- **source_model** (CompoundParameterModel, *optional*) – a model to lazily get data from

set_model()

class spinetoolbox.spine_db_editor.widgets.custom_qwidgets.**OpenFileButton**(*file_path*, *db_editor*)

Bases: PySide2.QtWidgets.QToolButton

A button to open files or show them in the folder.

open_file(*checked=False*)

open_containing_folder(*checked=False*)

class spinetoolbox.spine_db_editor.widgets.custom_qwidgets.**OpenSQLiteFileButton**(*file_path*, *db_editor*)

Bases: *OpenFileButton*

A button to open sqlite files, show them in the folder, or add them to the project.

open_file(*checked=False*)

class spinetoolbox.spine_db_editor.widgets.custom_qwidgets.**ShootingLabel**(*origin*, *destination*, *parent=None*, *duration=1200*)

Bases: PySide2.QtWidgets.QLabel

_handle_value_changed(*value*)

show()

spinetoolbox.spine_db_editor.widgets.edit_or_remove_items_dialogs

Classes for custom QDialogs to edit items in databases.

author

M. Marin (KTH)

date

13.5.2018

Module Contents

Classes

EditOrRemoveItemsDialog	A dialog with a CopyPasteTableView and a QDialogButtonBox. Base class for all
EditObjectClassesDialog	A dialog to query user's preferences for updating object classes.
EditObjectsDialog	A dialog to query user's preferences for updating objects.
EditRelationshipClassesDialog	A dialog to query user's preferences for updating relationship classes.
EditRelationshipsDialog	A dialog to query user's preferences for updating relationships.
RemoveEntitiesDialog	A dialog to query user's preferences for removing tree items.

class spinetoolbox.spine_db_editor.widgets.edit_or_remove_items_dialogs.**EditOrRemoveItemsDialog**(*parent*, *db_mngr*)

Bases: *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.ManageItemsDialog*

A dialog with a CopyPasteTableView and a QDialogButtonBox. Base class for all dialogs to query user's preferences for adding/editing/managing data items.

Init class.

Parameters

- **parent** (SpineDBEditor) – data store widget
- **db_mngr** (SpineDBManager) –

all_databases(*row*)

Returns a list of db names available for a given row. Used by delegates.

class spinetoolbox.spine_db_editor.widgets.edit_or_remove_items_dialogs.**EditObjectClassesDialog**(*parent*, *db_mngr*, *selected*)

Bases: *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.ShowIconColorEditorMixin*, *EditOrRemoveItemsDialog*

A dialog to query user's preferences for updating object classes.

Init class.

Parameters

- **parent** (SpineDBEditor) – data store widget
- **db_mngr** (SpineDBManager) – the manager to do the update
- **selected** (*set*) – set of ObjectClassItem instances to edit

connect_signals()

Connect signals to slots.

all_db_maps(*row*)

Returns a list of db maps available for a given row. Used by ShowIconColorEditorMixin.

accept()

Collect info from dialog and try to update items.

class spinetoolbox.spine_db_editor.widgets.edit_or_remove_items_dialogs.**EditObjectsDialog**(*parent*, *db_mngr*, *selected*)

Bases: *EditOrRemoveItemsDialog*

A dialog to query user's preferences for updating objects.

Init class.

Parameters

- **parent** (SpineDBEditor) – data store widget
- **db_mngr** (SpineDBManager) – the manager to do the update
- **selected** (*set*) – set of ObjectItem instances to edit

accept()

Collect info from dialog and try to update items.

class spinetoolbox.spine_db_editor.widgets.edit_or_remove_items_dialogs.**EditRelationshipClassesDialog**(*pa*
di
se
le

Bases: *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.ShowIconColorEditorMixin*, *EditOrRemoveItemsDialog*

A dialog to query user's preferences for updating relationship classes.

Init class.

Parameters

- **parent** (*SpineDBEditor*) – data store widget
- **db_mngr** (*SpineDBManager*) – the manager to do the update
- **selected** (*set*) – set of RelationshipClassItem instances to edit

connect_signals()

Connect signals to slots.

accept()

Collect info from dialog and try to update items.

class spinetoolbox.spine_db_editor.widgets.edit_or_remove_items_dialogs.**EditRelationshipsDialog**(*parent*, *db_mngr*, *selected*, *class_key*

Bases: *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.GetRelationshipClassesMixin*, *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.GetObjectsMixin*, *EditOrRemoveItemsDialog*

A dialog to query user's preferences for updating relationships.

Init class.

Parameters

- **parent** (*SpineDBEditor*) – data store widget
- **db_mngr** (*SpineDBManager*) – the manager to do the update
- **selected** (*set*) – set of RelationshipItem instances to edit
- **class_key** (*tuple*) – (class_name, object_class_name_list) for identifying the relationship_class

accept()

Collect info from dialog and try to update items.

class spinetoolbox.spine_db_editor.widgets.edit_or_remove_items_dialogs.**RemoveEntitiesDialog**(*parent*, *db_mngr*, *selected*)

Bases: *EditOrRemoveItemsDialog*

A dialog to query user's preferences for removing tree items.

Init class.

Parameters

- **parent** (SpineDBEditor) – data store widget
- **db_mngr** (SpineDBManager) – the manager to do the removal
- **selected** (*dict*) – maps item type (class) to instances

accept()

Collect info from dialog and try to remove items.

spinetoolbox.spine_db_editor.widgets.graph_layout_generator

Contains the GraphLayoutGeneratorRunnable class.

author

M. Marin (KTH)

date

26.11.2018

Module Contents

Classes

ProgressBarWidget	
GraphLayoutGeneratorRunnable	Computes the layout for the Entity Graph View.

Functions

make_heat_map(x, y, values)	

spinetoolbox.spine_db_editor.widgets.graph_layout_generator.**make_heat_map**(*x*, *y*, *values*)

class spinetoolbox.spine_db_editor.widgets.graph_layout_generator.**ProgressBarWidget**

Bases: PySide2.QtWidgets.QWidget

set_layout_generator(*layout_generator*)

paintEvent(*event*)

class spinetoolbox.spine_db_editor.widgets.graph_layout_generator.**GraphLayoutGeneratorRunnable**(*identifier*, *vertex_count*, *src_inds=(*, *dst_inds=(*, *spread=0*, *heavy_pos*, *max_iters=*, *weight_exp*, *2*)

Bases: PySide2.QtCore.QRunnable, spinedb_api.graph_layout_generator.GraphLayoutGenerator

Computes the layout for the Entity Graph View.

class Signals

Bases: PySide2.QtCore.QObject

finished

layout_available

progressed

msg

stop(*_checked=False*)

set_show_previews(*checked*)

emit_progressed(*iteration*)

emit_msg(*text*)

save_layout(*x*, *y*)

run()

spinetoolbox.spine_db_editor.widgets.graph_view_mixin

Contains the GraphViewMixin class.

author

M. Marin (KTH)

date

26.11.2018

Module Contents

Classes

GraphViewMixin	Provides the graph view for the DS form.

class spinetoolbox.spine_db_editor.widgets.graph_view_mixin.**GraphViewMixin**(**args*, ***kwargs*)

Provides the graph view for the DS form.

VERTEX_EXTENT = 64

_ARC_WIDTH

_ARC_LENGTH_HINT

_stop_extending_graph(*_=False*)

init_models()

connect_signals()

Connects signals.

receive_objects_added(*db_map_data*)

Runs when objects are added to the db. Adds the new objects to the graph if needed.

Parameters
db_map_data (*dict*) – list of dictionary-items keyed by DiffDatabaseMapping instance.

receive_relationships_added(*db_map_data*)

Runs when relationships are added to the db. Adds the new relationships to the graph if needed.

Parameters
db_map_data (*dict*) – list of dictionary-items keyed by DiffDatabaseMapping instance.

receive_object_classes_updated(*db_map_data*)

receive_relationship_classes_updated(*db_map_data*)

receive_objects_updated(*db_map_data*)

Runs when objects are updated in the db. Refreshes names of objects in graph.

Parameters
db_map_data (*dict*) – list of dictionary-items keyed by DiffDatabaseMapping instance.

receive_objects_removed(*db_map_data*)

Runs when objects are removed from the db. Rebuilds graph if needed.

Parameters
db_map_data (*dict*) – list of dictionary-items keyed by DiffDatabaseMapping instance.

receive_relationships_updated(*db_map_data*)

Runs when relationships are updated in the db.

Parameters
db_map_data (*dict*) – list of dictionary-items keyed by DiffDatabaseMapping instance.

receive_relationships_removed(*db_map_data*)

Runs when relationships are removed from the db. Rebuilds graph if needed.

Parameters
db_map_data (*dict*) – list of dictionary-items keyed by DiffDatabaseMapping instance.

add_db_map_ids_to_items(*db_map_data*, *type_*)

Goes through items of given type and adds the corresponding db_map ids. This could mean either restoring removed (db_map, id) tuples previously removed, or adding new (db_map, id) tuples.

Parameters
db_map_data (*dict(DiffDatabaseMapping, list)*) – List of added items keyed by db_map

Returns
tuples (db_map, id) that didn't match any item in the view.

Return type
list

hide_removed_entities(*db_map_data*, *type_*)

Hides removed entities while saving them into a set. This allows entities to be restored in case the user undoes the operation.

refresh_icons(*db_map_data*)

Runs when entity classes are updated in the db. Refreshes icons of entities in graph.

Parameters
db_map_data (*dict*) – list of dictionary-items keyed by DiffDatabaseMapping instance.

_handle_entity_graph_visibility_changed(*visible*)

rebuild_graph(*selected=None*)

Stores the given selection of entity tree indexes and builds graph.

build_graph(*persistent=False*)

Builds the graph.

Parameters
persistent (*bool, optional*) – If True, elements in the current graph (if any) retain their position in the new one.

_stop_layout_generators()

_complete_graph(*layout_gen_id*, *x*, *y*)

Parameters

- **layout_gen_id** (*object*) –
- **x** (*list*) – Horizontal coordinates
- **y** (*list*) – Vertical coordinates

_get_selected_entity_ids()

Returns a set of ids corresponding to selected entities in the trees.

Returns
selected object ids set: selected relationship ids

Return type
set

_get_db_map_relationships_for_graph(*db_map_object_ids*, *db_map_relationship_ids*)

_update_graph_data()

Updates data for graph according to selection in trees.

_get_object_key(*db_map_object_id*)

_get_relationship_key(*db_map_relationship_id*)

_update_src_dst_inds(*db_map_object_id_lists*)

_get_parameter_positions(*parameter_name*)

_make_layout_generator()

Returns a layout generator for the current graph.

Returns
GraphLayoutGeneratorRunnable

_make_new_items(*x*, *y*)

Returns new items for the graph.

Parameters

- **x** (*list*) –
- **y** (*list*) –

_add_new_items()

start_relationship(*db_map*, *relationship_class*, *obj_item*)

Starts a relationship from the given object item.

Parameters

- **db_map** (*DiffDatabaseMapping*) –
- **relationship_class** (*dict*) –
- **obj_item** (*..graphics_items.ObjectItem*) –

finalize_relationship(*relationship_class*, **object_items*)

Tries to add relationships between the given object items.

Parameters

- **relationship_class** (*dict*) –
- **object_items** (*..graphics_items.ObjectItem*) –

_begin_add_relationships()

_end_add_relationships()

add_objects_at_position(*pos*)

get_pdf_file_path()

closeEvent(*event*)

Handle close window.

Parameters
event (*QCloseEvent*) – Closing event

spinetoolbox.spine_db_editor.widgets.item_metadata_editor

Contains machinery to deal with item metadata editor.

author

A. Soininen (VTT)

date

25.3.2022

Module Contents

Classes

ItemMetadataEditor	A DB editor helper class that manages entity and parameter value metadata editor.

class spinetoolbox.spine_db_editor.widgets.item_metadata_editor.**ItemMetadataEditor**(*item_metadata_table_view*, *db_editor*, *metadata_editor*, *db_mngr*)

A DB editor helper class that manages entity and parameter value metadata editor.

Parameters

- **item_metadata_table_view** (ItemMetadataTableView) – editor's view
- **db_editor** (SpineDBEditor) – database editor
- **metadata_editor** (MetadataEditor) – metadata editor
- **db_mngr** (SpineDBManager) – database manager

connect_signals(*ui*)

Connects user interface signals.

Parameters

ui (Ui_MainWindow) – DB editor's user interface

init_models(*db_maps*)

Initializes editor's models.

Parameters

db_maps (*Iterable of DiffDatabaseMapping*) – database mappings

_cache_item_metadata(*db_maps*)

Caches item metadata into DB manager's cache.

Parameters

db_maps (*Iterable of DiffDatabaseMapping*) – database mappings

_reload_entity_metadata(*current_index*, *previous_index*)

Loads entity metadata for selected object or relationship.

Parameters

- **current_index** (*QModelIndex*) – currently selected index in object/relationship tree

- **previous_index** (*QModelIndex*) – unused

_reload_value_metadata(*current_index*, *previous_index*)

Loads parameter value metadata for selected value.

Parameters

- **current_index** (*QModelIndex*) – currently selected index in object/relationship parameter value table
- **previous_index** (*QModelIndex*) – unused

add_item_metadata(*db_map_data*)

Adds new item metadata records to the model and updates metadata model if required.

Parameters
db_map_data (*dict*) – added records keyed by database mapping

update_item_metadata(*db_map_data*)

Updates item metadata.

Parameters
db_map_data (*dict*) – updated metadata records

remove_item_metadata(*db_map_data*)

Removes item metadata records from the model.

Parameters
db_map_data (*dict*) – added records keyed by database mapping

update_metadata(*db_map_data*)

Updates metadata.

Parameters
db_map_data (*dict*) – updated metadata records

remove_metadata(*db_map_data*)

Removes entries corresponding to removed metadata from the model.

Parameters
db_map_data (*dict*) – removed metadata records

roll_back(*db_maps*)

Rolls back database changes.

Parameters
db_maps (*Iterable of DiffDatabaseMapping*) – rolled back databases

spinetoolbox.spine_db_editor.widgets.manage_items_dialogs

Classes for custom QDialogs to add edit and remove database items.

author

M. Marin (KTH)

date
13.5.2018

Module Contents

Classes

ManageItemsDialogBase	Init class.
ManageItemsDialog	A dialog with a CopyPasteTableView and a QDialogButtonBox. Base class for all
GetObjectClassesMixin	Provides a method to retrieve object classes for AddObjectsDialog and AddRelationshipClassesDialog.
GetObjectsMixin	Provides a method to retrieve objects for AddRelationshipsDialog and EditRelationshipsDialog.
GetRelationshipClassesMixin	Provides a method to retrieve relationship classes for AddRelationshipsDialog and EditRelationshipsDialog.
ShowIconColorEditorMixin	Provides methods to show an *IconColorEditor* upon request.

class spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.**ManageItemsDialogBase**(*parent*, *db_mngr*)

Bases: PySide2.QtWidgets.QDialog

Init class.

Parameters

- **parent** (SpineDBEditor) – data store widget
- **db_mngr** (SpineDBManager) –

make_table_view()

connect_signals()

Connect signals to slots.

resize_window_to_columns(*height=None*)

class spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.**ManageItemsDialog**(*parent*, *db_mngr*)

Bases: *ManageItemsDialogBase*

A dialog with a CopyPasteTableView and a QDialogButtonBox. Base class for all dialogs to query user's preferences for adding/editing/managing data items.

Init class.

Parameters

- **parent** (SpineDBEditor) – data store widget
- **db_mngr** (SpineDBManager) –

connect_signals()

Connect signals to slots.

_handle_model_data_changed(*top_left*, *bottom_right*, *roles*)

Reimplement in subclasses to handle changes in model data.

set_model_data(*index*, *data*)

Update model data.

_handle_model_reset()

Resize columns and form.

class spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.**GetObjectClassesMixin**

Provides a method to retrieve object classes for AddObjectsDialog and AddRelationshipClassesDialog.

make_db_map_obj_cls_lookup()

object_class_name_list(*row*)

Return a list of object_class names present in all databases selected for given row. Used by *ManageObjectsDelegate*.

class spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.**GetObjectsMixin**

Provides a method to retrieve objects for AddRelationshipsDialog and EditRelationshipsDialog.

make_db_map_obj_lookup()

object_name_list(*row*, *column*)

Return a list of object names present in all databases selected for given row. Used by *ManageRelationshipsDelegate*.

class spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.**GetRelationshipClassesMixin**

Provides a method to retrieve relationship classes for AddRelationshipsDialog and EditRelationshipsDialog.

make_db_map_rel_cls_lookup()

class spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.**ShowIconColorEditorMixin**

Provides methods to show an *IconColorEditor* upon request.

show_icon_color_editor(*index*)

spinetoolbox.spine_db_editor.widgets.mass_select_items_dialogs

Classes for custom QDialogs to add edit and remove database items.

author

M. Marin (KTH)

date

13.5.2018

Module Contents

Classes

MassSelectItemsDialog	A dialog to query a selection of dbs and items from the user.
MassRemoveItemsDialog	A dialog to query user's preferences for mass removing db items.
MassExportItemsDialog	A dialog to let users chose items for JSON export.

class spinetoolbox.spine_db_editor.widgets.mass_select_items_dialogs.**MassSelectItemsDialog**(*parent*, *db_mngr*, **db_maps*, *stored_state=No*

Bases: PySide2.QtWidgets.QDialog

A dialog to query a selection of dbs and items from the user.

Parameters

- **parent** (SpineDBEditor) – parent widget
- **db_mngr** (SpineDBManager) – database manager
- ***db_maps** – the dbs to select items from
- **stored_state** (*dict, Optional*) – widget's previous state

state_storing_requested

_handle_check_box_state_changed(*_checked*)

accept()

class spinetoolbox.spine_db_editor.widgets.mass_select_items_dialogs.**MassRemoveItemsDialog**(*parent*, *db_mngr*, **db_maps*, *stored_state=No*

Bases: *MassSelectItemsDialog*

A dialog to query user's preferences for mass removing db items.

Initialize class.

Parameters

- **parent** (SpineDBEditor) –
- **db_mngr** (SpineDBManager) –
- **db_maps** (*DiffDatabaseMapping*) – the dbs to select items from
- **stored_state** (*dict, Optional*) – widget's previous state

accept()

class spinetoolbox.spine_db_editor.widgets.mass_select_items_dialogs.**MassExportItemsDialog**(*parent*, *db_mngr*, **db_maps*, *stored_state=No*

Bases: *MassSelectItemsDialog*

A dialog to let users chose items for JSON export.

Parameters

- **parent** (SpineDBEditor) –
- **db_mngr** (SpineDBManager) –
- **db_maps** (*DiffDatabaseMapping*) – the dbs to select items from
- **stored_state** (*dict, Optional*) – widget's previous state

data_submitted

accept()

spinetoolbox.spine_db_editor.widgets.metadata_editor

Contains machinery to deal with metadata editor.

author

A. Soininen (VTT)

date

7.2.2022

Module Contents

Classes

MetadataEditor	A DB editor helper class that manages metadata editor.

class spinetoolbox.spine_db_editor.widgets.metadata_editor.**MetadataEditor**(*metadata_table_view*, *db_editor*, *db_mngr*)

A DB editor helper class that manages metadata editor.

Parameters

- **metadata_table_view** (MetadataTableView) – editor's view
- **db_editor** (SpineDBEditor) – database editor
- **db_mngr** (SpineDBManager) – database manager

connect_signals(*ui*)

Connects user interface signals.

Parameters

ui (Ui_MainWindow) – DB editor's user interface

init_models(*db_maps*)

Initializes editor's models.

Parameters

db_maps (*Iterable of DiffDatabaseMapping*) – database mappings

metadata_model()

Returns metadata model.

Returns

model

Return type

MetadataModel

add_metadata(*db_map_data*)

Adds metadata.

Parameters
db_map_data (*dict*) – added metadata records

update_metadata(*db_map_data*)

Updates metadata.

Parameters
db_map_data (*dict*) – updated metadata records

remove_metadata(*db_map_data*)

Removes entries corresponding to removed metadata from the model.

Parameters
db_map_data (*dict*) – removed metadata records

add_and_update_metadata(*db_map_data*)

Adds and updates metadata.

Combined metadata additions and updates may happen when item metadata has been updated.

Parameters
db_map_data (*dict*) – removed metadata records

roll_back(*db_maps*)

Rolls back database changes.

Parameters
db_maps (*Iterable of DiffDatabaseMapping*) – rolled back databases

spinetoolbox.spine_db_editor.widgets.multi_spine_db_editor

Contains the MultiSpineDBEditor class.

author

M. Marin (KTH)

date

12.12.2020

Module Contents

Classes

MultiSpineDBEditor	Database editor's tabbed main window.
_CustomStatusBar	

class spinetoolbox.spine_db_editor.widgets.multi_spine_db_editor.**MultiSpineDBEditor**(*db_mngr*, *db_url_codenames=None*

Bases: *spinetoolbox.widgets.multi_tab_window.MultiTabWindow*

Database editor's tabbed main window.

Parameters

- **db_mngr** (SpineDBManager) – database manager
- **db_url_codenames** (*dict, optional*) – mapping from database URL to its codename

_make_other()

Creates a new MultiTabWindow of this type.

Returns
new MultiTabWindow

Return type
MultiTabWindow

others()

List of other MultiTabWindows of the same type.

Returns
other MutliTabWindows windows

Return type
list of *MultiTabWindow*

_connect_tab_signals(*tab*)

Connects signals from a tab contents widget.

Parameters
tab (*QWidget*) – tab contents widget

Returns
True if signals were connected successfully, False otherwise

Return type
bool

_disconnect_tab_signals(*index*)

Disconnects signals from given tab.

Parameters
index (*int*) – tab index

Returns
True if signals were disconnected successfully, False otherwise

Return type
bool

_make_new_tab(*db_url_codenames=None*)

Creates a new tab.

Parameters

- ***args** – positional arguments neede to make a new tab
- ****kwargs** – keyword arguments needed to make a new tab

show_plus_button_context_menu(*global_pos*)

Opens a context menu for the tool bar.

Parameters
global_pos (*QPoint*) – menu position on screen

make_context_menu(*index*)

Creates a context menu for given tab.

Parameters
index (*int*) – tab index

Returns
context menu or None if tab was not found

Return type
QMenu

_insert_statusbar_button(*button*)

Inserts given button to the 'beginning' of the status bar and decorates it with a shooting label.

Parameters
button (OpenFileButton) –

insert_sqlite_file_open_button(*file_path*)

insert_file_open_button(*file_path*)

_open_sqlite_url(*url*, *codename*)

Opens sqlite url.

show_user_guide(*checked=False*)

Opens Spine db editor documentation page in browser.

_show_waiting_for_fetcher()

Shows a message box to user informing that a tab is waiting for fetcher to finish working.

_close_waiting_for_fetcher()

class spinetoolbox.spine_db_editor.widgets.multi_spine_db_editor.**_CustomStatusBar**(*parent=None*)

Bases: PySide2.QtWidgets.QStatusBar

spinetoolbox.spine_db_editor.widgets.object_name_list_editor

Contains the ObjectNameListEditor class.

author

M. Marin (KTH)

date
27.11.2019

Module Contents

Classes

SearchBarDelegate	A custom delegate to use with ObjectNameListEditor.
ObjectNameListEditor	A dialog to select the object name list for a relationship using Google-like search bars.

class spinetoolbox.spine_db_editor.widgets.object_name_list_editor.**SearchBarDelegate**

Bases: PySide2.QtWidgets.QItemDelegate

A custom delegate to use with ObjectNameListEditor.

data_committed

setModelData(*editor*, *model*, *index*)

createEditor(*parent*, *option*, *index*)

updateEditorGeometry(*editor*, *option*, *index*)

close_editor(*editor*, *index*, *model*)

eventFilter(*editor*, *event*)

class spinetoolbox.spine_db_editor.widgets.object_name_list_editor.**ObjectNameListEditor**(*parent*, *index*, *object_class_names*, *object_names_lists*, *current_object_names*)

Bases: *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.ManageItemsDialog*

A dialog to select the object name list for a relationship using Google-like search bars.

Initializes widget.

Parameters

- **parent** (SpineDBEditor) –
- **index** (*QModelIndex*) –
- **object_class_names** (*list*) – string object_class names
- **object_names_lists** (*list*) – lists of string object names
- **current_object_names** (*list*) –

init_model(*object_class_names*, *object_names_lists*, *current_object_names*)

accept()

spinetoolbox.spine_db_editor.widgets.parameter_view_mixin

Contains the ParameterViewMixin class.

author

M. Marin (KTH)

date

26.11.2018

Module Contents

Classes

ParameterViewMixin	Provides stacked parameter tables for the Spine db editor.

class spinetoolbox.spine_db_editor.widgets.parameter_view_mixin.**ParameterViewMixin**(**args*, ***kwargs*)

Provides stacked parameter tables for the Spine db editor.

connect_signals()

Connects signals to slots.

init_models()

Initializes models.

show_object_name_list_editor(*index*, *rel_cls_id*, *db_map*)

Shows the object names list editor.

Parameters

- **index** (*QModelIndex*) –
- **rel_cls_id** (*int*) –
- **db_map** (*DiffDatabaseMapping*) –

_set_default_parameter_data(*index=None*)

Sets default rows for parameter models according to given index.

Parameters

index (*QModelIndex*) – and index of the object or relationship tree

set_default_parameter_data(*default_data*, *default_db_map*)

clear_all_filters()

_reset_filters()

Resets filters.

_handle_graph_selection_changed(*selected_items*)

Resets filter according to graph selection.

_handle_object_tree_selection_changed(*selected_indexes*)

Resets filter according to object tree selection.

_handle_relationship_tree_selection_changed(*selected_indexes*)

Resets filter according to relationship tree selection.

_handle_alternative_selection_changed(*selected_db_map_alt_ids*)

Resets filter according to selection in alternative tree view.

receive_alternatives_updated(*db_map_data*)

receive_parameter_definitions_added(*db_map_data*)

receive_parameter_values_added(*db_map_data*)

receive_parameter_definitions_updated(*db_map_data*)

receive_parameter_values_updated(*db_map_data*)

receive_object_classes_removed(*db_map_data*)

receive_relationship_classes_removed(*db_map_data*)

receive_parameter_definitions_removed(*db_map_data*)

receive_parameter_values_removed(*db_map_data*)

spinetoolbox.spine_db_editor.widgets.pivot_table_header_view

Contains custom QHeaderView for the pivot table.

author

M. Marin (KTH)

date

2.12.2019

Module Contents

Classes

PivotTableHeaderView	Header view for the pivot table.
ParameterValuePivotHeaderView	Header view for the pivot table in parameter value and index expansion mode.
ScenarioAlternativePivotHeaderView	Header view for the pivot table in parameter value and index expansion mode.

class spinetoolbox.spine_db_editor.widgets.pivot_table_header_view.**PivotTableHeaderView**(*orientation*, *area*, *pivot_table_view*)

Bases: PySide2.QtWidgets.QHeaderView

Header view for the pivot table.

Parameters

- **orientation** (*int*) – Qt.Horizontal or Qt.Vertical
- **area** (*str*) – which pivot area the header represents: "columns", "rows" or "frozen"
- **pivot_table_view** (PivotTableView) – parent view

property area

header_dropped

dragEnterEvent(*event*)

dragMoveEvent(*event*)

dropEvent(*event*)

class spinetoolbox.spine_db_editor.widgets.pivot_table_header_view.**ParameterValuePivotHeaderView**(*orientati area, pivot_ta*

Bases: *PivotTableHeaderView*

Header view for the pivot table in parameter value and index expansion mode.

Parameters

- **orientation** (*int*) – Qt.Horizontal or Qt.Vertical
- **area** (*str*) – which pivot area the header represents: "columns", "rows" or "frozen"
- **pivot_table_view** (*PivotTableView*) – parent view

_column_selection()

Lists current column's indexes that contain some data.

Returns
column indexes

Return type
list of QModelIndex

_add_column_to_plot(*action*)

Adds a single column to existing plot window.

_plot_column()

Plots a single column not the selection.

_column_indexes(*column*)

Makes indexes for given column.

Parameters
column (*int*) – column

Returns
column indexes

Return type
list of QModelIndex

_set_x_flag()

Sets the X flag for a column.

contextMenuEvent(*event*)

Shows context menu.

Parameters
event (*QContextMenuEvent*) –

class spinetoolbox.spine_db_editor.widgets.pivot_table_header_view.**ScenarioAlternativePivotHeaderView**(*o a p*

Bases: *PivotTableHeaderView*

Header view for the pivot table in parameter value and index expansion mode.

Parameters

- **orientation** (*int*) – Qt.Horizontal or Qt.Vertical

- **area** (*str*) – which pivot area the header represents: "columns", "rows" or "frozen"
- **pivot_table_view** (PivotTableView) – parent view

context_menu_requested

Requests a header context menu be shown at given global position.

contextMenuEvent(*event*)

spinetoolbox.spine_db_editor.widgets.scenario_generator

Contains a dialog for generating scenarios from selected alternatives.

authors
A.Soininen (VTT)

date
7.9.2021

Module Contents

Classes

_ScenarioNameResolution	Generic enumeration.
ScenarioGenerator	A dialog where users can generate scenarios from given alternatives.

Functions

_ensure_unique(scenario_alternatives)	Removes duplicate scenario alternatives.
_find_base_alternative(names)	Returns the name of a 'base' alternative or empty string if not found.

class spinetoolbox.spine_db_editor.widgets.scenario_generator.**_ScenarioNameResolution**

Bases: enum.Enum

Generic enumeration.

Derive from this class to define new enumerations.

NO_CONFLICT

OVERWRITE

LEAVE_AS_IS

CANCEL_OPERATION

class spinetoolbox.spine_db_editor.widgets.scenario_generator.**ScenarioGenerator**(*parent*, *db_map*, *alternatives*, *spine_db_editor*)

Bases: `PySide2.QtWidgets.QWidget`

A dialog where users can generate scenarios from given alternatives.

Parameters

- **parent** (*QWidget*) – parent widget
- **db_map** (*DiffDatabaseMapping*) – database mapping that contains the alternatives
- **alternatives** (*Iterable of CacheItem*) – alternatives from which the scenarios are generated
- **spine_db_editor** (SpineDBEditor) – database editor instance

_TYPE_LABELS = ['All combinations', 'Scenario for each alternative']

accept()

Generates scenarios and closes the dialog.

The operation may get cancelled by user if there are conflicts in scenario names.

_generate_scenarios(*new_scenarios*, *scenarios_to_modify*, *scenario_alternatives*)

Generates scenarios with all possible combinations of given alternatives.

Parameters

- **new_scenarios** (*Iterable of str*) – names of new scenarios to create
- **scenarios_to_modify** (*Iterable of str*) – names of scenarios to modify
- **scenario_alternatives** (*list of list*) – alternative items for each scenario

_check_existing_scenarios(*proposed_scenario_names*, *existing_scenario_names*)

Checks if proposed scenarios exist, and if so, prompts users what to do.

Parameters

- **proposed_scenario_names** (*Iterable of str*) – proposed scenario names
- **existing_scenario_names** (*set of str*) – existing scenario names

Returns
action to take

Return type
_ScenarioNameResolution

_enable_base_alternative(*check_box_state*)

Enables and disables base alternative combo box.

Parameters
check_box_state (*int*) – state of 'Use base alternative' check box

_insert_base_alternative(*scenario_alternatives*)

Prepends base alternative to scenario alternatives if it has been enabled.

If base alternative is already in scenario alternatives, make sure it comes first.

Parameters
scenario_alternatives (*list of list*) – scenario alternatives

spinetoolbox.spine_db_editor.widgets.scenario_generator.**_ensure_unique**(*scenario_alternatives*)

Removes duplicate scenario alternatives.

Parameters
scenario_alternatives (*list of list*) – scenario alternatives

spinetoolbox.spine_db_editor.widgets.scenario_generator.**_find_base_alternative**(*names*)

Returns the name of a 'base' alternative or empty string if not found.

Basically, checks if "Base" is in names, otherwise searches for the first case-insensitive version of "base".

Parameters
names (*list of str*) – alternative names

Returns
base alternative name

Return type
str

spinetoolbox.spine_db_editor.widgets.select_position_parameters_dialog

Classes for custom QDialogs to add items to databases.

author

M. Marin (KTH)

date
13.5.2018

Module Contents

Classes

SelectPositionParametersDialog	
ParameterNameDelegate	A delegate for the database name.

class spinetoolbox.spine_db_editor.widgets.select_position_parameters_dialog.**SelectPositionParametersDi**

Bases: PySide2.QtWidgets.QDialog

selection_made

accept()

_parameter_position_x()

_parameter_position_y()

class spinetoolbox.spine_db_editor.widgets.select_position_parameters_dialog.**ParameterNameDelegate**(*paren*
db_m
**db_*

Bases: PySide2.QtWidgets.QStyledItemDelegate

A delegate for the database name.

setModelData(*editor*, *model*, *index*)

Send signal.

setEditorData(*editor*, *index*)

Do nothing. We're setting editor data right away in createEditor.

updateEditorGeometry(*editor*, *option*, *index*)

_close_editor(*editor*, *index*)

Closes editor. Needed by SearchBarEditor.

createEditor(*parent*, *option*, *index*)

Returns editor.

spinetoolbox.spine_db_editor.widgets.spine_db_editor

Contains the SpineDBEditor class.

author

M. Marin (KTH)

date

26.11.2018

Module Contents

Classes

SpineDBEditorBase	Base class for SpineDBEditor (i.e. Spine database editor).
SpineDBEditor	A widget to visualize Spine dbs.

class spinetoolbox.spine_db_editor.widgets.spine_db_editor.**SpineDBEditorBase**(*db_mngr*)

Bases: PySide2.QtWidgets.QMainWindow

Base class for SpineDBEditor (i.e. Spine database editor).

Parameters

db_mngr (SpineDBManager) – The manager to use

property toolbox

property settings_subgroup

property db_names

property first_db_map

property db_url_codenames

msg

msg_error

file_exported

sqlite_file_exported

static is_db_map_editor()

Always returns True as SpineDBEditors are truly database editors.

Unless, of course, the database can one day be opened in read-only mode. In that case this method should return False.

Returns
Always True

Return type
bool

load_db_urls(*db_url_codenames*, *create=False*, *update_history=True*)

init_add_undo_redo_actions()

load_previous_urls(*_=False*)

load_next_urls(*_=False*)

open_db_file(*_=False*)

add_db_file(*_=False*)

create_db_file(*_=False*)

_make_docks_menu()

Returns a menu with all dock toggle/view actions. Called by `self.add_main_menu()`.

Returns
QMenu

add_main_menu()

Adds a menu with main actions to toolbar.

_browse_commits()

connect_signals()

Connects signals to slots.

vacuum(*_checked=False*)

update_undo_redo_actions(*index*)

_replace_undo_redo_actions(*new_undo_action*, *new_redo_action*)

_refresh_undo_redo_actions()

update_commit_enabled(*_clean=False*)

init_models()

Initializes models.

add_message(*msg*)

Pushes message to notification stack.

Parameters

msg (*str*) – String to show in the notification

refresh_copy_paste_actions()

Runs when menus are about to show. Enables or disables actions according to selection status.

copy(*checked=False*)

Copies data to clipboard.

paste(*checked=False*)

Pastes data from clipboard.

import_data(*data*)

import_file(*checked=False*)

Import file. It supports SQLite, JSON, and Excel.

import_from_json(*file_path*)

import_from_sqlite(*file_path*)

import_from_excel(*file_path*)

show_mass_export_items_dialog(*checked=False*)

Shows dialog for user to select dbs and items for export.

_store_export_settings(*state*)

Stores export items dialog settings.

_clean_up_export_items_dialog()

Cleans up export items dialog.

export_session(*checked=False*)

Exports changes made in the current session as reported by DiffDatabaseMapping.

mass_export_items(*db_map_item_types*)

duplicate_object(*object_item*)

Duplicates an object.

Parameters

object_item (*ObjectTreeItem of* ObjectItem) –

duplicate_scenario(*db_map*, *scen_id*)

Duplicates a scenario.

Parameters

- **db_map** (*DiffDatabaseMapping*) –
- **scen_id** (*int*) –

export_data(*db_map_ids_for_export*)

Exports data from given dictionary into a file.

Parameters

db_map_ids_for_export – Dictionary mapping db maps to keyword arguments for spinedb_api.export_data

refresh_session(*checked=False*)

commit_session(*checked=False*)

Commits dirty database maps.

rollback_session(*checked=False*)

Rolls back dirty database maps.

receive_session_committed(*db_maps*, *cookie*)

receive_session_rolled_back(*db_maps*)

receive_session_refreshed(*db_maps*)

show_mass_remove_items_form(*checked=False*)

Opens the purge items dialog.

_store_purge_settings(*state*)

Stores Purge items dialog state.

Parameters

state (*dict*) – dialog state

_clean_up_purge_items_dialog()

Removes references to purge items dialog.

show_parameter_value_editor(*index*, *plain=False*)

Shows the parameter_value editor for the given index of given table view.

receive_error_msg(*db_map_error_log*)

_update_export_enabled()

Update export enabled.

_receive_items_changed(*action*, *item_type*, *db_map_data*)

Enables or disables actions and informs the user about what just happened.

receive_scenarios_added(*db_map_data*)

receive_alternatives_added(*db_map_data*)

receive_object_classes_added(*db_map_data*)

receive_objects_added(*db_map_data*)

receive_relationship_classes_added(*db_map_data*)

receive_relationships_added(*db_map_data*)

receive_entity_groups_added(*db_map_data*)

receive_parameter_definitions_added(*db_map_data*)

receive_parameter_values_added(*db_map_data*)

receive_parameter_value_lists_added(*db_map_data*)

receive_list_values_added(*db_map_data*)

receive_features_added(*db_map_data*)

receive_tools_added(*db_map_data*)

receive_tool_features_added(*db_map_data*)

receive_tool_feature_methods_added(*db_map_data*)

receive_metadata_added(*db_map_data*)

receive_entity_metadata_added(*db_map_data*)

receive_parameter_value_metadata_added(*db_map_data*)

receive_scenarios_updated(*db_map_data*)

receive_alternatives_updated(*db_map_data*)

receive_object_classes_updated(*db_map_data*)

receive_objects_updated(*db_map_data*)

receive_relationship_classes_updated(*db_map_data*)

receive_relationships_updated(*db_map_data*)

receive_parameter_definitions_updated(*db_map_data*)

receive_parameter_values_updated(*db_map_data*)

receive_parameter_value_lists_updated(*db_map_data*)

receive_list_values_updated(*db_map_data*)

receive_features_updated(*db_map_data*)

receive_tools_updated(*db_map_data*)

receive_tool_features_updated(*db_map_data*)

receive_tool_feature_methods_updated(*db_map_data*)

receive_metadata_updated(*db_map_data*)

receive_entity_metadata_updated(*db_map_data*)

receive_parameter_value_metadata_updated(*db_map_data*)

receive_scenarios_removed(*db_map_data*)

receive_alternatives_removed(*db_map_data*)

receive_object_classes_removed(*db_map_data*)

receive_objects_removed(*db_map_data*)

receive_relationship_classes_removed(*db_map_data*)

receive_relationships_removed(*db_map_data*)

receive_entity_groups_removed(*db_map_data*)

receive_parameter_definitions_removed(*db_map_data*)

receive_parameter_values_removed(*db_map_data*)

receive_parameter_value_lists_removed(*db_map_data*)

receive_list_values_removed(*db_map_data*)

receive_features_removed(*db_map_data*)

receive_tools_removed(*db_map_data*)

receive_tool_features_removed(*db_map_data*)

receive_tool_feature_methods_removed(*db_map_data*)

receive_metadata_removed(*db_map_data*)

receive_entity_metadata_removed(*db_map_data*)

receive_parameter_value_metadata_removed(*db_map_data*)

restore_ui()

Restore UI state from previous session.

save_window_state()

Save window state parameters (size, position, state) via QSettings.

tear_down()

Performs clean up duties.

Returns
True if editor is ready to close, False otherwise

Return type
bool

_prompt_to_commit_changes()

Prompts the user to commit or rollback changes to 'dirty' db maps.

Returns
QMessageBox status code

Return type
int

_get_commit_msg(*db_names*)

Prompts user for commit message.

Parameters
db_names (*Iterable of str*) – database names

Returns
commit message

Return type
str

`_get_rollback_confirmation`(*db_names*)

Prompts user for confirmation before rolling back the session.

Parameters
db_names (*Iterable of str*) – database names

Returns
True if user confirmed, False otherwise

Return type
bool

`closeEvent`(*event*)

Handle close window.

Parameters
event (*QCloseEvent*) – Closing event

`scenario_items`(*db_map*)

Gathers scenario items from alternative scenario tree for given database.

Parameters
db_map (*DiffDatabaseMapping*) – database map

Returns
scenario items

Return type
list of CachedItem

class spinetoolbox.spine_db_editor.widgets.spine_db_editor.**SpineDBEditor**(*db_mngr*, *db_url_codenames=None*)

Bases: *spinetoolbox.spine_db_editor.widgets.tabular_view_mixin.TabularViewMixin*, *spinetoolbox.spine_db_editor.widgets.graph_view_mixin.GraphViewMixin*, *spinetoolbox.spine_db_editor.widgets.parameter_view_mixin.ParameterViewMixin*, *spinetoolbox.spine_db_editor.widgets.tree_view_mixin.TreeViewMixin*, *SpineDBEditorBase*

A widget to visualize Spine dbs.

Initializes everything.

Parameters
db_mngr (*SpineDBManager*) – The manager to use

`pinned_values_updated`

`emit_pinned_values_updated`()

`connect_signals`()

Connects signals to slots.

`init_models`()

Initializes models.

`_restart_timer_refresh_tab_order`(*_visible=False*)

`_refresh_tab_order`()

tabify_and_raise(*docks*)

Tabifies docks in given list, then raises the first.

Parameters
docks (*list*) –

restore_dock_widgets()

Docks all floating and or hidden QDockWidgets back to the window.

begin_style_change()

Begins a style change operation.

end_style_change()

Ends a style change operation.

apply_stacked_style(*checked=False*)

Applies the stacked style, inspired in the former tree view.

apply_pivot_style(*_action*)

Applies the pivot style, inspired in the former tabular view.

apply_graph_style(*checked=False*)

Applies the graph style, inspired in the former graph view.

receive_metadata_added(*db_map_data*)

receive_entity_metadata_added(*db_map_data*)

receive_parameter_value_metadata_added(*db_map_data*)

receive_metadata_updated(*db_map_data*)

receive_entity_metadata_updated(*db_map_data*)

receive_parameter_value_metadata_updated(*db_map_data*)

receive_metadata_removed(*db_map_data*)

receive_entity_metadata_removed(*db_map_data*)

receive_parameter_value_metadata_removed(*db_map_data*)

receive_session_rolled_back(*db_maps*)

Reacts to session rolled back event.

tear_down()

Performs clean up duties.

Returns
True if editor is ready to close, False otherwise

Return type
bool

static _get_base_dir()

spinetoolbox.spine_db_editor.widgets.tabular_view_header_widget

Contains TabularViewHeaderWidget class.

authors

P. Vennström (VTT), M. Marin (KTH)

date

2.12.2019

Module Contents

Classes

TabularViewHeaderWidget	A draggable QWidget.

class spinetoolbox.spine_db_editor.widgets.tabular_view_header_widget.**TabularViewHeaderWidget**(*identifier*, *area*, *menu=None*, *parent=None*)

Bases: PySide2.QtWidgets.QFrame

A draggable QWidget.

Parameters

- **identifier** (*str*) –
- **area** (*str*) – either "rows", "columns", or "frozen"
- **menu** (*FilterMenu, optional*) –
- **parent** (*QWidget, optional*) – Parent widget

property identifier

property area

header_dropped

_H_MARGIN = 3

_SPACING = 16

mousePressEvent(*event*)

Register drag start position

mouseMoveEvent(*event*)

Start dragging action if needed

mouseReleaseEvent(*event*)

Forget drag start position

dragEnterEvent(*event*)

dropEvent(*event*)

spinetoolbox.spine_db_editor.widgets.tabular_view_mixin

Contains TabularViewMixin class.

author

P. Vennström (VTT)

date

1.11.2018

Module Contents

Classes

TabularViewMixin	Provides the pivot table and its frozen table for the Database editor.

class spinetoolbox.spine_db_editor.widgets.tabular_view_mixin.**TabularViewMixin**(**args*, ***kwargs*)

Provides the pivot table and its frozen table for the Database editor.

property current_object_class_id_list

property current_object_class_name_list

property current_object_class_ids

_PARAMETER_VALUE = &Value

_INDEX_EXPANSION = &Index

_RELATIONSHIP = Re&lationship

_SCENARIO_ALTERNATIVE = &Scenario

_PARAMETER = parameter

_ALTERNATIVE = alternative

_INDEX = index

populate_pivot_action_group()

connect_signals()

Connects signals to slots.

_connect_pivot_table_header_signals()

Connects signals of pivot table's header views.

init_models()

Initializes models.

_set_model_data(*index*, *value*)

static _is_class_index(*index*)

Returns whether or not the given tree index is a class index.

Parameters
index (*QModelIndex*) – index from object or relationship tree

Returns
bool

_handle_pivot_action_triggered(*action*)

_handle_pivot_table_visibility_changed(*visible*)

_handle_frozen_table_visibility_changed(*visible*)

_handle_object_tree_selection_changed(*selected_indexes*)

_handle_relationship_tree_selection_changed(*selected_indexes*)

_handle_entity_tree_current_changed(*current_index*)

_update_class_attributes(*current_index*)

Updates current class (type and id) and reloads pivot table for it.

static _get_current_class_item(*current_index*)

static _make_get_id(*action*)

Returns a function to compute the db_map-id tuple of an item.

_get_db_map_entities()

Returns a dict mapping db maps to a list of dict entity items in the current class.

Returns
dict

load_empty_relationship_data(*db_map_class_objects=None*)

Returns a dict containing all possible relationships in the current class.

Parameters
db_map_class_objects (*dict*) –

Returns
Key is db_map-object_id tuple, value is None.

Return type
dict

load_full_relationship_data(*db_map_relationships=None*, *action='add'*)

Returns a dict of relationships in the current class.

Parameters
db_map_relationships (*dict*) –

Returns
Key is db_map-object id tuple, value is relationship id.

Return type
dict

load_relationship_data()

Returns a dict that merges empty and full relationship data.

Returns
Key is object id tuple, value is True if a relationship exists, False otherwise.

Return type
dict

load_scenario_alternative_data(*db_map_scenarios=None*, *db_map_alternatives=None*)

Returns a dict containing all scenario alternatives.

Returns
Key is db_map-id tuple, value is None or rank.

Return type
dict

_get_db_map_parameter_value_or_def_ids(*item_type*)

Returns a dict mapping db maps to a list of integer parameter (value or def) ids from the current class.

Parameters
item_type (*str*) – either "parameter_value" or "parameter_definition"

Returns
dict

_get_db_map_parameter_values_or_defs(*item_type*)

Returns a dict mapping db maps to list of dict parameter (value or def) items from the current class.

Parameters
item_type (*str*) – either "parameter_value" or "parameter_definition"

Returns
dict

load_empty_parameter_value_data(*db_map_entities=None*, *db_map_parameter_ids=None*, *db_map_alternative_ids=None*)

Returns a dict containing all possible combinations of entities and parameters for the current class in all db_maps.

Parameters

- **db_map_entities** (*dict, optional*) – if given, only load data for these db maps and entities
- **db_map_parameter_ids** (*dict, optional*) – if given, only load data for these db maps and parameter definitions
- **db_map_alternative_ids** (*dict, optional*) – if given, only load data for these db maps and alternatives

Returns
Key is a tuple object_id, . . . , parameter_id, value is None.

Return type
dict

load_full_parameter_value_data(*db_map_parameter_values=None*, *action='add'*)

Returns a dict of parameter values for the current class.

Parameters

- **db_map_parameter_values** (*list, optional*) –
- **action** (*str*) –

Returns
Key is a tuple object_id, ..., parameter_id, value is the parameter_value.

Return type
dict

_indexes(*value*)

load_empty_expanded_parameter_value_data(*db_map_entities=None*, *db_map_parameter_ids=None*, *db_map_alternative_ids=None*)

Makes a dict of expanded parameter values for the current class.

Parameters

- **db_map_parameter_values** (*list, optional*) –
- **action** (*str*) –

Returns
mapping from unique value id tuple to value tuple

Return type
dict

load_full_expanded_parameter_value_data(*db_map_parameter_values=None*, *action='add'*)

Makes a dict of expanded parameter values for the current class.

Parameters

- **db_map_parameter_values** (*list, optional*) –
- **action** (*str*) –

Returns
mapping from unique value id tuple to value tuple

Return type
dict

load_parameter_value_data()

Returns a dict that merges empty and full parameter_value data.

Returns
Key is a tuple object_id, ..., parameter_id, value is the parameter_value or None if not specified.

Return type
dict

load_expanded_parameter_value_data()

Returns all permutations of entities as well as parameter indexes and values for the current class.

Returns
Key is a tuple object_id, ..., index, while value is None.

Return type
dict

get_pivot_preferences()

Returns saved pivot preferences.

Returns
pivot tuple, or None if no preference stored

Return type
tuple, NoneType

do_reload_pivot_table()

Reloads pivot table.

_can_build_pivot_table()

clear_pivot_table()

wipe_out_filter_menus()

make_pivot_headers()

Turns top left indexes in the pivot table into TabularViewHeaderWidget.

_resize_pivot_header_columns()

make_frozen_headers()

Turns indexes in the first row of the frozen table into TabularViewHeaderWidget.

create_filter_menu(*identifier*)

Returns a filter menu for given given object_class identifier.

Parameters
identifier (*int*) –

Returns
TabularViewFilterMenu

create_header_widget(*identifier*, *area*, *with_menu=True*)

Returns a TabularViewHeaderWidget for given object_class identifier.

Parameters

- **identifier** (*str*) –
- **area** (*str*) –
- **with_menu** (*bool*) –

Returns
TabularViewHeaderWidget

static _get_insert_index(*pivot_list*, *catcher*, *position*)

Returns an index for inserting a new element in the given pivot list.

Returns
int

handle_header_dropped(*dropped*, *catcher*, *position=''*)

Updates pivots when a header is dropped.

Parameters

- **dropped** (TabularViewHeaderWidget) –

- **catcher** (TabularViewHeaderWidget, PivotTableHeaderView, FrozenTableView) –
- **position** (*str*) – either "before", "after", or ""

get_frozen_value(*index*)

Returns the value in the frozen table corresponding to the given index.

Parameters
index (*QModelIndex*) –

Returns
tuple

change_frozen_value(*current*, *previous*)

Sets the frozen value from selection in frozen table.

change_filter(*identifier*, *valid_values*, *has_filter*)

reload_frozen_table()

Resets the frozen model according to new selection in entity trees.

find_frozen_values(*frozen*)

Returns a list of tuples containing unique values (object ids) for the frozen indexes (object_class ids).

Parameters
frozen (*tuple(int)*) – A tuple of currently frozen indexes

Returns
list(tuple(list(int)))

static refresh_table_view(*table_view*)

update_filter_menus(*action*)

receive_objects_added_or_removed(*db_map_data*, *action*)

receive_relationships_added_or_removed(*db_map_data*, *action*)

receive_parameter_definitions_added_or_removed(*db_map_data*, *action*)

receive_alternatives_added_or_removed(*db_map_data*, *action*)

receive_parameter_values_added_or_removed(*db_map_data*, *action*)

receive_scenarios_added_or_removed(*db_map_data*, *action*)

receive_db_map_data_updated(*db_map_data*, *get_class_id*)

receive_classes_updated(*db_map_data*)

receive_classes_removed(*db_map_data*)

receive_alternatives_added(*db_map_data*)

Reacts to alternatives added event.

receive_scenarios_added(*db_map_data*)

Reacts to scenarios added event.

receive_objects_added(*db_map_data*)

Reacts to objects added event.

receive_relationships_added(*db_map_data*)

Reacts to relationships added event.

receive_parameter_definitions_added(*db_map_data*)

Reacts to parameter definitions added event.

receive_parameter_values_added(*db_map_data*)

Reacts to parameter values added event.

receive_alternatives_updated(*db_map_data*)

Reacts to alternatives updated event.

receive_object_classes_updated(*db_map_data*)

Reacts to object classes updated event.

receive_relationship_classes_updated(*db_map_data*)

Reacts to relationship classes updated event.

receive_objects_updated(*db_map_data*)

Reacts to objects updated event.

receive_relationships_updated(*db_map_data*)

Reacts to relationships updated event.

receive_parameter_values_updated(*db_map_data*)

Reacts to parameter values added event.

receive_parameter_definitions_updated(*db_map_data*)

Reacts to parameter definitions updated event.

receive_scenarios_updated(*db_map_data*)

receive_alternatives_removed(*db_map_data*)

Reacts to alternatives removed event.

receive_scenarios_removed(*db_map_data*)

Reacts to scenarios removed event.

receive_object_classes_removed(*db_map_data*)

Reacts to object classes removed event.

receive_relationship_classes_removed(*db_map_data*)

Reacts to relationship classes remove event.

receive_objects_removed(*db_map_data*)

Reacts to objects removed event.

receive_relationships_removed(*db_map_data*)

Reacts to relationships removed event.

receive_parameter_definitions_removed(*db_map_data*)

Reacts to parameter definitions removed event.

receive_parameter_values_removed(*db_map_data*)

Reacts to parameter values removed event.

receive_session_rolled_back(*db_maps*)

Reacts to session rolled back event.

spinetoolbox.spine_db_editor.widgets.tree_view_mixin

Contains the TreeViewMixin class.

author

M. Marin (KTH)

date

26.11.2018

Module Contents

Classes

TreeViewMixin	Provides object and relationship trees for the Spine db editor.

class spinetoolbox.spine_db_editor.widgets.tree_view_mixin.**TreeViewMixin**(**args*, ***kwargs*)

Provides object and relationship trees for the Spine db editor.

connect_signals()

Connects signals to slots.

init_models()

Initializes models.

static _db_map_items(*indexes*)

Groups items from given tree indexes by db map.

Returns

lists of dictionary items keyed by DiffDatabaseMapping

Return type

dict

_db_map_ids(*indexes*)

_db_map_class_ids(*indexes*)

export_selected(*selected_indexes*)

Exports data from given indexes in the entity tree.

show_add_object_classes_form()

Shows dialog to add new object classes.

show_add_objects_form(*parent_item*)

Shows dialog to add new objects.

show_add_object_group_form(*object_class_item*)

Shows dialog to add new object group.

show_manage_members_form(*object_item*)

Shows dialog to manage an object group.

show_add_relationship_classes_form(*parent_item*)

Shows dialog to add new relationship_class.

show_add_relationships_form(*parent_item*)

Shows dialog to add new relationships.

show_manage_relationships_form(*parent_item*)

edit_entity_tree_items(*selected_indexes*)

Starts editing given indexes.

show_edit_object_classes_form(*items*)

show_edit_objects_form(*items*)

show_edit_relationship_classes_form(*items*)

show_edit_relationships_form(*items*)

remove_entity_tree_items(*selected_indexes*)

Shows form to remove items from object treeview.

show_remove_entity_tree_items_form(*selected*)

receive_alternatives_added(*db_map_data*)

receive_scenarios_added(*db_map_data*)

receive_object_classes_added(*db_map_data*)

receive_objects_added(*db_map_data*)

receive_relationship_classes_added(*db_map_data*)

receive_relationships_added(*db_map_data*)

receive_entity_groups_added(*db_map_data*)

receive_parameter_value_lists_added(*db_map_data*)

receive_list_values_added(*db_map_data*)

receive_features_added(*db_map_data*)

receive_tools_added(*db_map_data*)

receive_tool_features_added(*db_map_data*)

receive_tool_feature_methods_added(*db_map_data*)

receive_alternatives_updated(*db_map_data*)

receive_scenarios_updated(*db_map_data*)

receive_object_classes_updated(*db_map_data*)

receive_objects_updated(*db_map_data*)

receive_relationship_classes_updated(*db_map_data*)

receive_relationships_updated(*db_map_data*)

receive_parameter_value_lists_updated(*db_map_data*)

receive_list_values_updated(*db_map_data*)

receive_features_updated(*db_map_data*)

receive_tools_updated(*db_map_data*)

receive_tool_features_updated(*db_map_data*)

receive_tool_feature_methods_updated(*db_map_data*)

receive_alternatives_removed(*db_map_data*)

receive_scenarios_removed(*db_map_data*)

receive_object_classes_removed(*db_map_data*)

receive_objects_removed(*db_map_data*)

receive_relationship_classes_removed(*db_map_data*)

receive_relationships_removed(*db_map_data*)

receive_entity_groups_removed(*db_map_data*)

receive_parameter_value_lists_removed(*db_map_data*)

receive_list_values_removed(*db_map_data*)

receive_features_removed(*db_map_data*)

receive_tools_removed(*db_map_data*)

receive_tool_features_removed(*db_map_data*)

receive_tool_feature_methods_removed(*db_map_data*)

spinetoolbox.spine_db_editor.widgets.url_toolbar

Contains the UrlToolBar class and helpers.

author

M. Marin (KTH)

date

13.5.2020

Module Contents

Classes

UrlToolBar
_FilterWidget
_FilterArrayWidget
_DBListWidget
_UrlFilterDialog

class spinetoolbox.spine_db_editor.widgets.url_toolbar.**UrlToolBar**(*db_editor*)

Bases: PySide2.QtWidgets.QToolBar

property line_edit

_add_open_project_url_menu()

_update_ds_url_menu_enabled()

_connect_project_item_model_signals(*slot*)

_disconnect_project_item_model_signals(*slot*)

_update_open_project_url_menu()

_open_ds_url(*action*)

add_main_menu(*menu*)

_update_history_actions_availability()

add_urls_to_history(*db_urls*)

Adds url to history.

Parameters
db_urls (*list of str*) –

get_previous_urls()

Returns previous urls in history.

Returns
list of str

get_next_urls()

Returns next urls in history.

Returns
list of str

_handle_line_edit_return_pressed()

set_current_urls(*urls*)

_show_filter_menu(*_checked=False*)

class spinetoolbox.spine_db_editor.widgets.url_toolbar.**_FilterWidget**(*db_mngr*, *db_map*, *item_type*, *filter_type*, *active_item*, *parent=None*)

Bases: PySide2.QtWidgets.QTreeWidget

sizeHint()

filter_config()

class spinetoolbox.spine_db_editor.widgets.url_toolbar.**_FilterArrayWidget**(*db_mngr*, *db_map*, *parent=None*)

Bases: PySide2.QtWidgets.QWidget

filter_selection_changed

filtered_url_codename()

sizeHint()

moveEvent(*ev*)

class spinetoolbox.spine_db_editor.widgets.url_toolbar.**_DBListWidget**(*db_mngr*, *db_maps*, *parent=None*)

Bases: PySide2.QtWidgets.QTreeWidget

db_filter_selection_changed

sizeHint()

filtered_url_codenames()

class spinetoolbox.spine_db_editor.widgets.url_toolbar.**_UrlFilterDialog**(*db_mngr*, *db_maps*, *parent=None*)

Bases: PySide2.QtWidgets.QDialog

filter_accepted

sizeHint()

_update_filter_enabled()

accept()

Submodules

spinetoolbox.spine_db_editor.graphics_items

Classes for drawing graphics items on graph view's QGraphicsScene.

authors

M. Marin (KTH), P. Savolainen (VTT)

date

4.4.2018

Module Contents

Classes

EntityItem	Base class for ObjectItem and RelationshipItem.
RelationshipItem	Represents a relationship in the Entity graph.
ObjectItem	Represents an object in the Entity graph.
ArcItem	Connects a RelationshipItem to an ObjectItem.
CrossHairsItem	Creates new relationships directly in the graph.
CrossHairsRelationshipItem	Represents the relationship that's being created using the CrossHairsItem.
CrossHairsArcItem	Connects a CrossHairsRelationshipItem with the CrossHairsItem,
ObjectLabelItem	Provides a label for ObjectItem's.

Functions

make_figure_graphics_item(scene[, z, static])	Creates a FigureCanvas and adds it to the given scene.

spinetoolbox.spine_db_editor.graphics_items.**make_figure_graphics_item**(*scene*, *z=0*, *static=True*)

Creates a FigureCanvas and adds it to the given scene. Used for creating heatmaps and associated colorbars.

Parameters

- **scene** (*QGraphicsScene*) –
- **z** (*int, optional*) – z value. Defaults to 0.
- **static** (*bool, optional*) – if True (the default) the figure canvas is not movable

Returns
the graphics item that represents the canvas Figure: the figure in the canvas

Return type
QGraphicsProxyWidget

class spinetoolbox.spine_db_editor.graphics_items.**EntityItem**(*spine_db_editor*, *x*, *y*, *extent*, *db_map_ids*)

Bases: PySide2.QtWidgets.QGraphicsRectItem

Base class for ObjectItem and RelationshipItem.

Parameters

- **spine_db_editor** (SpineDBEditor) – 'owner'
- **x** (*float*) – x-coordinate of central point
- **y** (*float*) – y-coordinate of central point
- **extent** (*int*) – Preferred extent
- **db_map_ids** (*tuple*) – tuple of (db_map, id) tuples

abstract property entity_type

property db_map_ids

property original_db_map_ids

property entity_class_type

property entity_name

property first_entity_class_id

property entity_class_name

property first_db_map_id

property first_id

property first_db_map

property display_data

property display_database

property db_maps

abstract _make_tool_tip()

abstract default_parameter_data()

entity_class_id(*db_map*)

entity_id(*db_map*)

db_map_data(*db_map*)

db_map_id(*db_map*)

boundingRect()

moveBy(*dx*, *dy*)

_init_bg()

refresh_icon()

Refreshes the icon.

_set_renderer(*renderer*)

shape()

Returns a shape containing the entire bounding rect, to work better with icon transparency.

paint(*painter*, *option*, *widget=None*)

Shows or hides the selection halo.

_paint_as_selected()

_paint_as_deselected()

add_arc_item(*arc_item*)

Adds an item to the list of arcs.

Parameters

arc_item (ArcItem) –

apply_zoom(*factor*)

Applies zoom.

Parameters
factor (*float*) – The zoom factor.

apply_rotation(*angle*, *center*)

Applies rotation.

Parameters

- **angle** (*float*) – The angle in degrees.
- **center** (*QPointF*) – Rotates around this point.

block_move_by(*dx*, *dy*)

mouseMoveEvent(*event*)

Moves the item and all connected arcs.

Parameters
event (*QGraphicsSceneMouseEvent*) –

update_arcs_line()

Moves arc items.

itemChange(*change*, *value*)

Keeps track of item's movements on the scene.

Parameters

- **change** (*GraphicsItemChange*) – a flag signalling the type of the change
- **value** – a value related to the change

Returns
the same value given as input

setVisible(*on*)

Sets visibility status for this item and all arc items.

Parameters
on (*bool*) –

_make_menu()

contextMenuEvent(*e*)

Shows context menu.

Parameters
e (*QGraphicsSceneMouseEvent*) – Mouse event

remove_db_map_ids(*db_map_ids*)

Removes db_map_ids.

add_db_map_ids(*db_map_ids*)

class spinetoolbox.spine_db_editor.graphics_items.**RelationshipItem**(*spine_db_editor*, *x*, *y*, *extent*, *db_map_ids*)

Bases: *EntityItem*

Represents a relationship in the Entity graph.

Initializes the item.

Parameters

- **spine_db_editor** (*GraphViewForm*) – 'owner'
- **x** (*float*) – x-coordinate of central point
- **y** (*float*) – y-coordinate of central point
- **extent** (*int*) – preferred extent
- **db_map_ids** (*tuple*) – tuple of (db_map, id) tuples

property entity_type

property object_class_id_list

property object_name_list

default_parameter_data()

Return data to put as default in a parameter table when this item is selected.

object_id_list(*db_map*)

db_representation(*db_map*)

_make_tool_tip()

_init_bg()

follow_object_by(*dx*, *dy*)

add_arc_item(*arc_item*)

Adds an item to the list of arcs.

Parameters
arc_item (ArcItem) –

itemChange(*change*, *value*)

Rotates svg item if the relationship is 2D. This makes it possible to define e.g. an arow icon for relationships that express direction.

_rotate_svg_item()

class spinetoolbox.spine_db_editor.graphics_items.**ObjectItem**(*spine_db_editor*, *x*, *y*, *extent*, *db_map_ids*)

Bases: *EntityItem*

Represents an object in the Entity graph.

Initializes the item.

Parameters

- **spine_db_editor** (*GraphViewForm*) – 'owner'
- **x** (*float*) – x-coordinate of central point
- **y** (*float*) – y-coordinate of central point
- **extent** (*int*) – preferred extent
- **db_map_ids** (*tuple*) – tuple of (db_map, id) tuples

property entity_type

default_parameter_data()

Return data to put as default in a parameter table when this item is selected.

db_representation(*db_map*)

shape()

Returns a shape containing the entire bounding rect, to work better with icon transparency.

update_name()

Refreshes the name.

_make_tool_tip()

block_move_by(*dx*, *dy*)

mouseDoubleClickEvent(*e*)

_make_menu()

_duplicate()

_refresh_relationship_classes()

_populate_expand_collapse_menu(*menu*)

Populates the 'Expand' or 'Collapse' menu.

Parameters

menu (*QMenu*) –

_populate_add_relationships_menu(*menu*)

Populates the 'Add relationships' menu.

Parameters

menu (*QMenu*) –

_get_db_map_relationship_ids_to_expand_or_collapse(*action*)

_expand(*action*)

_collapse(*action*)

_start_relationship(*action*)

class spinetoolbox.spine_db_editor.graphics_items.**ArcItem**(*rel_item*, *obj_item*, *width*)

Bases: PySide2.QtWidgets.QGraphicsPathItem

Connects a RelationshipItem to an ObjectItem.

Initializes item.

Parameters

- **rel_item** (*spinetoolbox.widgets.graph_view_graphics_items.RelationshipItem*) – relationship item
- **obj_item** (*spinetoolbox.widgets.graph_view_graphics_items.ObjectItem*) – object item
- **width** (*float*) – Preferred line width

_make_pen()

moveBy(*dx*, *dy*)

Does nothing. This item is not moved the regular way, but follows the EntityItems it connects.

update_line()

mousePressEvent(*event*)

Accepts the event so it's not propagated.

other_item(*item*)

apply_zoom(*factor*)

Applies zoom.

Parameters

factor (*float*) – The zoom factor.

class spinetoolbox.spine_db_editor.graphics_items.**CrossHairsItem**(**args*, ***kwargs*)

Bases: *RelationshipItem*

Creates new relationships directly in the graph.

Initializes the item.

Parameters

- **spine_db_editor** (*GraphViewForm*) – 'owner'
- **x** (*float*) – x-coordinate of central point
- **y** (*float*) – y-coordinate of central point
- **extent** (*int*) – preferred extent
- **db_map_ids** (*tuple*) – tuple of (db_map, id) tuples

property entity_class_name

property entity_name

_make_tool_tip()

refresh_icon()

Refreshes the icon.

set_plus_icon()

set_check_icon()

set_normal_icon()

set_ban_icon()

set_icon(*unicode*, *color=0*)

Refreshes the icon.

mouseMoveEvent(*event*)

Moves the item and all connected arcs.

Parameters

event (*QGraphicsSceneMouseEvent*) –

block_move_by(*dx*, *dy*)

contextMenuEvent(*e*)

Shows context menu.

Parameters
e (*QGraphicsSceneMouseEvent*) – Mouse event

class spinetoolbox.spine_db_editor.graphics_items.**CrossHairsRelationshipItem**(**args*, ***kwargs*)

Bases: *RelationshipItem*

Represents the relationship that's being created using the CrossHairsItem.

Initializes the item.

Parameters

- **spine_db_editor** (*GraphViewForm*) – 'owner'
- **x** (*float*) – x-coordinate of central point
- **y** (*float*) – y-coordinate of central point
- **extent** (*int*) – preferred extent
- **db_map_ids** (*tuple*) – tuple of (db_map, id) tuples

_make_tool_tip()

refresh_icon()

Refreshes the icon.

contextMenuEvent(*e*)

Shows context menu.

Parameters
e (*QGraphicsSceneMouseEvent*) – Mouse event

class spinetoolbox.spine_db_editor.graphics_items.**CrossHairsArcItem**(*rel_item*, *obj_item*, *width*)

Bases: *ArcItem*

Connects a CrossHairsRelationshipItem with the CrossHairsItem, and with all the ObjectItem's in the relationship so far.

Initializes item.

Parameters

- **rel_item** (*spinetoolbox.widgets.graph_view_graphics_items.RelationshipItem*) – relationship item
- **obj_item** (*spinetoolbox.widgets.graph_view_graphics_items.ObjectItem*) – object item
- **width** (*float*) – Preferred line width

_make_pen()

class spinetoolbox.spine_db_editor.graphics_items.**ObjectLabelItem**(*entity_item*)

Bases: PySide2.QtWidgets.QGraphicsTextItem

Provides a label for ObjectItem's.

Initializes item.

Parameters
entity_item (*spinetoolbox.widgets.graph_view_graphics_items.EntityItem*) – The parent item.

entity_name_edited

setPlainText(*text*)

Set texts and resets position.

Parameters
text (*str*) –

reset_position()

Adapts item geometry so text is always centered.

spinetoolbox.spine_db_editor.main

Module Contents

Functions

main()	Launches Spine Db Editor as it's own application.
_make_argument_parser()	Builds a command line argument parser.

spinetoolbox.spine_db_editor.main.**main**()

Launches Spine Db Editor as it's own application.

spinetoolbox.spine_db_editor.main.**_make_argument_parser**()

Builds a command line argument parser.

Returns
parser

Return type
ArgumentParser

spinetoolbox.spine_db_editor.scenario_generation

Contains functions for automatically generating scenarios from a set of alternatives.

authors
A.Soininen (VTT)

date
7.9.2021

Module Contents

Functions

all_combinations(alternatives)	Creates all possible combinations of alternatives.
unique_alternatives(alternatives)	Creates all possible single-alternative scenarios.

spinetoolbox.spine_db_editor.scenario_generation.**all_combinations**(*alternatives*)

Creates all possible combinations of alternatives.

Parameters
alternatives (*Iterable of Any*) – alternatives

Returns
lists containing alternatives for each scenario

Return type
list of list

spinetoolbox.spine_db_editor.scenario_generation.**unique_alternatives**(*alternatives*)

Creates all possible single-alternative scenarios.

Parameters
alternatives (*Iterable of Any*) – alternatives

Returns
tuples containing alternatives for each scenario

Return type
list of list

spinetoolbox.widgets

Init file for widgets package. Intentionally empty.

author

P. Savolainen (VTT)

date
3.1.2018

Submodules

spinetoolbox.widgets.about_widget

A widget for presenting basic information about the application.

author

P. Savolainen (VTT)

date
14.12.2017

Module Contents

Classes

AboutWidget	About widget class.

class spinetoolbox.widgets.about_widget.**AboutWidget**(*toolbox*)

Bases: PySide2.QtWidgets.QWidget

About widget class.

Parameters
toolbox (ToolboxUI) – QMainWindow instance

calc_pos()

Calculate the top-left corner position of this widget in relation to main window position and size in order to show about window in the middle of the main window.

setup_license_text()

Add license to QTextBrowser.

keyPressEvent(*e*)

Close form when Escape, Enter, Return, or Space bar keys are pressed.

Parameters
e (*QKeyEvent*) – Received key press event.

closeEvent(*event=None*)

Handle close window.

Parameters
event (*QEvent*) – Closing event if 'X' is clicked.

mousePressEvent(*e*)

Save mouse position at the start of dragging.

Parameters
e (*QMouseEvent*) – Mouse event

mouseReleaseEvent(*e*)

Save mouse position at the end of dragging.

Parameters
e (*QMouseEvent*) – Mouse event

mouseMoveEvent(*e*)

Moves the window when mouse button is pressed and mouse cursor is moved.

Parameters
e (*QMouseEvent*) – Mouse event

spinetoolbox.widgets.add_project_item_widget

Widget shown to user when a new Project Item is created.

author

P. Savolainen (VTT)

date

19.1.2017

Module Contents

Classes

AddProjectItemWidget	A widget to query user's preferences for a new item.

class spinetoolbox.widgets.add_project_item_widget.**AddProjectItemWidget**(*toolbox*, *x*, *y*, *class_*, *spec=''*)

Bases: PySide2.QtWidgets.QWidget

A widget to query user's preferences for a new item.

toolbox

Parent widget

Type

ToolboxUI

x

X coordinate of new item

Type

int

y

Y coordinate of new item

Type

int

Initialize class.

connect_signals()

Connect signals to slots.

handle_name_changed()

Update label to show upcoming folder name.

handle_ok_clicked()

Check that given item name is valid and add it to project.

abstract call_add_item()

Creates new Item according to user's selections.

Must be reimplemented by subclasses.

keyPressEvent(*e*)

Close Setup form when escape key is pressed.

Parameters
e (*QKeyEvent*) – Received key press event.

closeEvent(*event=None*)

Handle close window.

Parameters
event (*QEvent*) – Closing event if 'X' is clicked.

spinetoolbox.widgets.add_up_spine_opt_wizard

Classes for custom QDialogs for julia setup.

author

M. Marin (KTH)

date
13.5.2018

Module Contents

Classes

_PageId	Enum where members are also (and must be) ints
AddUpSpineOptWizard	A wizard to install & upgrade SpineOpt.
IntroPage	
SelectJuliaPage	
CheckPreviousInstallPage	
AddUpSpineOptPage	A QWizards page with a log. Useful for pages that need to capture the output of a process.
SuccessPage	
FailurePage	
TroubleshootProblemsPage	
TroubleshootSolutionPage	
ResetRegistryPage	A QWizards page with a log. Useful for pages that need to capture the output of a process.
AddUpSpineOptAgainPage	A QWizards page with a log. Useful for pages that need to capture the output of a process.
TotalFailurePage	

Functions

_clear_layout(layout)

class spinetoolbox.widgets.add_up_spine_opt_wizard.**_PageId**

Bases: enum.IntEnum

Enum where members are also (and must be) ints

Initialize self. See help(type(self)) for accurate signature.

INTRO

SELECT_JULIA

CHECK_PREVIOUS_INSTALL

ADD_UP_SPINE_OPT

SUCCESS

FAILURE

TROUBLESHOOT_PROBLEMS

TROUBLESHOOT_SOLUTION

RESET_REGISTRY

ADD_UP_SPINE_OPT_AGAIN

TOTAL_FAILURE

class spinetoolbox.widgets.add_up_spine_opt_wizard.**AddUpSpineOptWizard**(*parent*, *julia_exe*, *julia_project*)

Bases: PySide2.QtWidgets.QWizard

A wizard to install & upgrade SpineOpt.

Initialize class.

Parameters
parent (*QWidget*) – the parent widget (SettingsWidget)

class spinetoolbox.widgets.add_up_spine_opt_wizard.**IntroPage**(*parent*)

Bases: PySide2.QtWidgets.QWizardPage

nextId()

class spinetoolbox.widgets.add_up_spine_opt_wizard.**SelectJuliaPage**(*parent*, *julia_exe*, *julia_project*)

Bases: PySide2.QtWidgets.QWizardPage

initializePage()

_select_julia_exe()

_select_julia_project()

nextId()

class spinetoolbox.widgets.add_up_spine_opt_wizard.**CheckPreviousInstallPage**(*parent*)

Bases: PySide2.QtWidgets.QWizardPage

isComplete()

cleanupPage()

initializePage()

_handle_check_install_finished(*ret*)

nextId()

class spinetoolbox.widgets.add_up_spine_opt_wizard.**AddUpSpineOptPage**(*parent*)

Bases: *spinetoolbox.widgets.custom_qwidgets.QWizardProcessPage*

A QWizards page with a log. Useful for pages that need to capture the output of a process.

initializePage()

_handle_spine_opt_add_up_finished(*ret*)

nextId()

class spinetoolbox.widgets.add_up_spine_opt_wizard.**SuccessPage**(*parent*)

Bases: PySide2.QtWidgets.QWizardPage

initializePage()

nextId()

class spinetoolbox.widgets.add_up_spine_opt_wizard.**FailurePage**(*parent*)

Bases: PySide2.QtWidgets.QWizardPage

_handle_check_box_clicked(*checked=False*)

initializePage()

nextId()

class spinetoolbox.widgets.add_up_spine_opt_wizard.**TroubleshootProblemsPage**(*parent*)

Bases: PySide2.QtWidgets.QWizardPage

isComplete()

_show_log(*_=False*)

nextId()

class spinetoolbox.widgets.add_up_spine_opt_wizard.**TroubleshootSolutionPage**(*parent*)

Bases: PySide2.QtWidgets.QWizardPage

cleanupPage()

initializePage()

_initialize_page_solution1()

_initialize_page_solution2()

nextId()

class spinetoolbox.widgets.add_up_spine_opt_wizard.**ResetRegistryPage**(*parent*)

Bases: *spinetoolbox.widgets.custom_qwidgets.QWizardProcessPage*

A QWizards page with a log. Useful for pages that need to capture the output of a process.

initializePage()

_handle_registry_reset_finished(*ret*)

nextId()

class spinetoolbox.widgets.add_up_spine_opt_wizard.**AddUpSpineOptAgainPage**(*parent*)

Bases: *AddUpSpineOptPage*

A QWizards page with a log. Useful for pages that need to capture the output of a process.

nextId()

class spinetoolbox.widgets.add_up_spine_opt_wizard.**TotalFailurePage**(*parent*)

Bases: PySide2.QtWidgets.QWizardPage

nextId()

spinetoolbox.widgets.add_up_spine_opt_wizard.**_clear_layout**(*layout*)

spinetoolbox.widgets.array_editor

Contains an editor widget for array type parameter values.

author

A. Soininen (VTT)

date

25.3.2020

Module Contents

Classes

ArrayEditor	Editor widget for Arrays.

class spinetoolbox.widgets.array_editor.**ArrayEditor**(*parent=None*)

Bases: PySide2.QtWidgets.QWidget

Editor widget for Arrays.

Parameters

parent (*QWidget, optional*) – parent widget

set_value(*value*)

Sets the parameter_value for editing in this widget.

Parameters
value (*Array*) – value for editing

value()

Returns the array currently being edited.

Returns
array

Return type
Array

_check_if_plotting_enabled(*type_name*)

Checks is array's data type allows the array to be plotted.

Parameters
type_name (*str*) – data type's name

_change_value_type(*type_name*)

open_value_editor(*index*)

Opens an editor widget for array element.

Parameters
index (*QModelIndex*) – element's index

_show_table_context_menu(*position*)

Shows the table's context menu.

Parameters
position (*QPoint*) – menu's position on the table

_update_plot(*topLeft=None*, *bottomRight=None*, *roles=None*)

Updates the plot widget.

_open_header_editor(*column*)

spinetoolbox.widgets.array_value_editor

An editor dialog for Array elements.

author

A. Soininen (VTT)

date
10.11.2020

Module Contents

Classes

ArrayValueEditor	Editor widget for Array elements.

class spinetoolbox.widgets.array_value_editor.**ArrayValueEditor**(*index*, *value_type*, *parent=None*)

Bases: *spinetoolbox.widgets.parameter_value_editor_base.ParameterValueEditorBase*

Editor widget for Array elements.

Parameters

- **index** (*QModelIndex*) – an index to a parameter_value in parent_model
- **parent** (*QWidget, optional*) – a parent widget

_set_data(*value*)

See base class.

spinetoolbox.widgets.code_text_edit

Provides simple text editor for programming purposes.

author

M. Marin (KTH)

date

28.1.2020

Module Contents

Classes

CodeTextEdit	A plain text edit with syntax highlighting and line numbers.
LineNumberArea	

class spinetoolbox.widgets.code_text_edit.**CodeTextEdit**(**arg*, ***kwargs*)

Bases: PySide2.QtWidgets.QPlainTextEdit

A plain text edit with syntax highlighting and line numbers.

insertFromMimeData(*source*)

set_lexer_name(*lexer_name*)

setPlainText(*text*)

setDocument(*doc*)

line_number_area_width()

_update_line_number_area_width(*_new_block_count=0*)

_update_line_number_area(*rect*, *dy*)

_update_line_number_area_cursor_position()

resizeEvent(*event*)

line_number_area_paint_event(*ev*)

class spinetoolbox.widgets.code_text_edit.**LineNumberArea**(*editor*)

Bases: PySide2.QtWidgets.QWidget

sizeHint()

paintEvent(*ev*)

spinetoolbox.widgets.commit_dialog

Classes for custom QDialogs to add edit and remove database items.

author

M. Marin (KTH)

date

13.5.2018

Module Contents

Classes

CommitDialog	A dialog to query user's preferences for new commit.

class spinetoolbox.widgets.commit_dialog.**CommitDialog**(*parent*, **db_names*)

Bases: PySide2.QtWidgets.QDialog

A dialog to query user's preferences for new commit.

Parameters

- **parent** (*QWidget*) – the parent widget
- **db_names** (*Iterable of str*) – database names

receive_text_changed()

Called when text changes in the commit msg text edit. Enable/disable commit button accordingly.

spinetoolbox.widgets.console_window

Window for the 'base' Julia Console and Python Console.

author

P. Savolainen (VTT)

date

5.2.2021

Module Contents

Classes

ConsoleWindow	Class for a separate window for the Python or Julia Console.

class spinetoolbox.widgets.console_window.**ConsoleWindow**(*toolbox*, *spine_console*, *language*)

Bases: PySide2.QtWidgets.QMainWindow

Class for a separate window for the Python or Julia Console.

Parameters

- **toolbox** (ToolboxUI) – QMainWindow instance
- **spine_console** (JupyterConsoleWidget) – Qt Console
- **language** (*str*) – 'python' or 'julia'

start()

Starts the kernel.

closeEvent(*e*)

Shuts down the running kernel and calls ToolboxUI method to destroy this window.

Parameters

e (*QCloseEvent*) – Event

spinetoolbox.widgets.custom_combobox

A widget for presenting basic information about the application.

author

P. Savolainen (VTT)

date

14.12.2017

Module Contents

Classes

ElidedCombobox	Combobox with elided text.
OpenProjectDialogComboBox	

class spinetoolbox.widgets.custom_combobox.**ElidedCombobox**

Bases: PySide2.QtWidgets.QComboBox

Combobox with elided text.

paintEvent(*event*)

class spinetoolbox.widgets.custom_combobox.**OpenProjectDialogComboBox**

Bases: PySide2.QtWidgets.QComboBox

keyPressEvent(*e*)

Interrupts Enter and Return key presses when QComboBox is in focus. This is needed to prevent showing the 'Not a valid Spine Toolbox project' Notifier every time Enter is pressed.

Parameters
e (*QKeyEvent*) – Received key press event.

spinetoolbox.widgets.custom_delegates

Custom item delegates.

author

M. Marin (KTH)

date
1.9.2018

Module Contents

Classes

ComboBoxDelegate	
CheckBoxDelegate	A delegate that places a fully functioning QCheckBox.
RankDelegate	A delegate that places a QCheckBox but draws a number instead of the check.

class spinetoolbox.widgets.custom_delegates.**ComboBoxDelegate**(*items*)

Bases: PySide2.QtWidgets.QStyledItemDelegate

createEditor(*parent*, *option*, *index*)

paint(*painter*, *option*, *index*)

setEditorData(*editor*, *index*)

setModelData(*editor*, *model*, *index*)

updateEditorGeometry(*editor*, *option*, *index*)

_finalize_editing(*editor*)

class spinetoolbox.widgets.custom_delegates.**CheckBoxDelegate**(*parent*, *centered=True*)

Bases: PySide2.QtWidgets.QStyledItemDelegate

A delegate that places a fully functioning QCheckBox.

Parameters

- **parent** (*QWiget*) –
- **centered** (*bool*) – whether or not the checkbox should be center-aligned in the widget

data_committed

createEditor(*parent*, *option*, *index*)

Important, otherwise an editor is created if the user clicks in this cell. ** Need to hook up a signal to the model.

paint(*painter*, *option*, *index*)

Paint a checkbox without the label.

static _do_paint(*painter*, *checkbox_style_option*, *index*)

editorEvent(*event*, *model*, *option*, *index*)

Change the data in the model and the state of the checkbox when user presses left mouse button and this cell is editable. Otherwise do nothing.

setModelData(*editor*, *model*, *index*)

Do nothing. Model data is updated by handling the *data_committed* signal.

get_checkbox_rect(*option*)

class spinetoolbox.widgets.custom_delegates.**RankDelegate**(*parent*, *centered=True*)

Bases: *CheckBoxDelegate*

A delegate that places a QCheckBox but draws a number instead of the check.

Parameters

- **parent** (*QWiget*) –
- **centered** (*bool*) – whether or not the checkbox should be center-aligned in the widget

static _do_paint(*painter*, *checkbox_style_option*, *index*)

spinetoolbox.widgets.custom_editors

Custom editors for model/view programming.

author

M. Marin (KTH)

date

2.9.2018

Module Contents

Classes

CustomLineEditor	A custom QLineEdit to handle data from models.
ParameterValueLineEditor	A custom QLineEdit to handle data from models.
_CustomLineEditDelegate	A delegate for placing a CustomLineEditor on the first row of SearchBarEditor.
SearchBarEditor	A Google-like search bar, implemented as a QTableView with a _CustomLineEditDelegate in the first row.
CheckListEditor	A check list editor.
_IconPainterDelegate	A delegate to highlight decorations in a QListWidget.
IconColorEditor	An editor to let the user select an icon and a color for an object_class.

class spinetoolbox.widgets.custom_editors.**CustomLineEditor**

Bases: PySide2.QtWidgets.QLineEdit

A custom QLineEdit to handle data from models.

set_data(*data*)

Sets editor's text.

Parameters

data (*Any*) – anything convertible to string

data()

Returns editor's text.

Returns

editor's text

Return type

str

keyPressEvent(*event*)

Prevents shift key press to clear the contents.

class spinetoolbox.widgets.custom_editors.**ParameterValueLineEditor**

Bases: *CustomLineEditor*

A custom QLineEdit to handle data from models.

set_data(*data*)

Sets editor's text.

Parameters
data (*Any*) – anything convertible to string

data()

Returns editor's text.

Returns
editor's text

Return type
str

class spinetoolbox.widgets.custom_editors.**_CustomLineEditDelegate**

Bases: PySide2.QtWidgets.QStyledItemDelegate

A delegate for placing a CustomLineEditor on the first row of SearchBarEditor.

text_edited

setModelData(*editor*, *model*, *index*)

createEditor(*parent*, *option*, *index*)

Create editor and 'forward' *textEdited* signal.

eventFilter(*editor*, *event*)

Handle all sort of special cases.

class spinetoolbox.widgets.custom_editors.**SearchBarEditor**(*parent*, *tutor=None*)

Bases: PySide2.QtWidgets.QTableView

A Google-like search bar, implemented as a QTableView with a _CustomLineEditDelegate in the first row.

Initializes instance.

Parameters

- **parent** (*QWidget*) – parent widget
- **tutor** (*QWidget, optional*) – another widget used for positioning.

data_committed

set_data(*current*, *items*)

Populates model.

Parameters

- **current** (*str*) – item that is currently selected from given items
- **items** (*Sequence(str)*) – items to show in the list

set_base_offset(*offset*)

update_geometry(*option*)

Updates geometry.

refit()

data()

_handle_delegate_text_edited(*text*)

Filters model as the first row is being edited.

_proxy_model_filter_accepts_row(*source_row*, *source_parent*)

Always accept first row.

keyPressEvent(*event*)

Sets data from current index into first index as the user navigates through the table using the up and down keys.

currentChanged(*current*, *previous*)

edit_first_index()

Edits first index if valid and not already being edited.

mouseMoveEvent(*event*)

Sets the current index to the one hovered by the mouse.

mousePressEvent(*event*)

Commits data.

class spinetoolbox.widgets.custom_editors.**CheckListEditor**(*parent*, *tutor=None*, *ranked=False*)

Bases: PySide2.QtWidgets.QTableView

A check list editor.

Initialize class.

_make_icon(*i=None*)

keyPressEvent(*event*)

Toggles checked state if the user presses space.

toggle_selected(*index*)

Adds or removes given index from selected items.

Parameters

index (*QModelIndex*) – index to toggle

_select_item(*qitem*, *rank*)

_deselect_item(*qitem*, *update_ranks=False*)

mouseMoveEvent(*event*)

Sets the current index to the one under mouse.

mousePressEvent(*event*)

Toggles checked state of pressed index.

set_data(*items*, *checked_items*)

Sets data and updates geometry.

Parameters

- **items** (*Sequence(str)*) – All items.
- **checked_items** (*Sequence(str)*) – Initially checked items.

data()

Returns a comma separated list of checked items.

Returns

str

update_geometry(*option*)

Updates geometry.

class spinetoolbox.widgets.custom_editors.**_IconPainterDelegate**

Bases: PySide2.QtWidgets.QStyledItemDelegate

A delegate to highlight decorations in a QListWidget.

paint(*painter*, *option*, *index*)

Paints selected items using the highlight brush.

class spinetoolbox.widgets.custom_editors.**IconColorEditor**(*parent*)

Bases: PySide2.QtWidgets.QDialog

An editor to let the user select an icon and a color for an object_class.

Init class.

_proxy_model_filter_accepts_row(*source_row*, *source_parent*)

Filters icons according to search terms.

connect_signals()

Connects signals to slots.

set_data(*data*)

data()

spinetoolbox.widgets.custom_menus

Classes for custom context menus and pop-up menus.

author

P. Savolainen (VTT)

date

9.1.2018

Module Contents

Classes

CustomContextMenu	Context menu master class for several context menus.
OpenProjectDialogComboBoxContextMenu	Context menu master class for several context menus.
CustomPopupMenu	Popup menu master class for several popup menus.
ItemSpecificationMenu	Context menu class for item specifications.
RecentProjectsPopupMenu	Recent projects menu embedded to 'File-Open recent' QAction.
FilterMenuBase	Filter menu.

class spinetoolbox.widgets.custom_menus.**CustomContextMenu**(*parent*, *position*)

Bases: PySide2.QtWidgets.QMenu

Context menu master class for several context menus.

Parameters

- **parent** (*QWidget*) – Parent for menu widget (ToolboxUI)
- **position** (*QPoint*) – Position on screen

add_action(*text*, *icon=QIcon()*, *enabled=True*)

Adds an action to the context menu.

Parameters

- **text** (*str*) – Text description of the action
- **icon** (*QIcon*) – Icon for menu item
- **enabled** (*bool*) – Is action enabled?

set_action(*option*)

Sets the action which was clicked.

Parameters
option (*str*) – string with the text description of the action

get_action()

Returns the clicked action, a string with a description.

class spinetoolbox.widgets.custom_menus.**OpenProjectDialogComboBoxContextMenu**(*parent*, *position*)

Bases: *CustomContextMenu*

Context menu master class for several context menus.

Parameters

- **parent** (*QWidget*) – Parent for menu widget
- **position** (*QPoint*) – Position on screen

class spinetoolbox.widgets.custom_menus.**CustomPopupMenu**(*parent*)

Bases: PySide2.QtWidgets.QMenu

Popup menu master class for several popup menus.

Parameters
parent (*QWidget*) – Parent widget of this pop-up menu

add_action(*text*, *slot*, *enabled=True*, *tooltip=None*)

Adds an action to the popup menu.

Parameters

- **text** (*str*) – Text description of the action
- **slot** (*method*) – Method to connect to action's triggered signal
- **enabled** (*bool*) – Is action enabled?
- **tooltip** (*str*) – Tool tip for the action

class spinetoolbox.widgets.custom_menus.**ItemSpecificationMenu**(*toolbox*, *index*, *item=None*)

Bases: *CustomPopupMenu*

Context menu class for item specifications.

Parameters

- **toolbox** (ToolboxUI) – Toolbox that requests this menu, used as parent.
- **index** (*QModelIndex*) – the index
- **item** (ProjectItem, *optional*) – passed to show_specification_form

class spinetoolbox.widgets.custom_menus.**RecentProjectsPopupMenu**(*parent*)

Bases: *CustomPopupMenu*

Recent projects menu embedded to 'File-Open recent' QAction.

Parameters
parent (*QWidget*) – Parent widget of this menu (ToolboxUI)

add_recent_projects()

Reads the previous project names and paths from QSettings. Adds them to the QMenu as QActions.

call_open_project(*checked*, *p*)

Slot for catching the user selected action from the recent projects menu.

Parameters

- **checked** (*bool*) – Argument sent by triggered signal
- **p** (*str*) – Full path to a project file

class spinetoolbox.widgets.custom_menus.**FilterMenuBase**(*parent*)

Bases: PySide2.QtWidgets.QMenu

Filter menu.

Parameters
parent (*QWidget*) – a parent widget

connect_signals()

set_filter_list(*data*)

add_items_to_filter_list(*items*)

remove_items_from_filter_list(*items*)

_clear_filter()

_check_filter()

_change_filter()

abstract emit_filter_changed(*valid_values*)

wipe_out()

spinetoolbox.widgets.custom_qcombobox

Class for a custom QComboBox.

author

P. Savolainen (VTT)

date

16.10.2020

Module Contents

Classes

CustomQComboBox	A custom QComboBox for showing kernels in Settings->Tools.

class spinetoolbox.widgets.custom_qcombobox.**CustomQComboBox**

Bases: PySide2.QtWidgets.QComboBox

A custom QComboBox for showing kernels in Settings->Tools.

mouseMoveEvent(*e*)

Catch mouseMoveEvent and accept it because the comboBox popup (QListView) has mouse tracking on as default. This makes sure the comboBox popup appears in correct position and clicking on the combobox repeatedly does not move the Settings window.

spinetoolbox.widgets.custom_qgraphicsscene

Custom QGraphicsScene used in the Design View.

author

P. Savolainen (VTT)

date

13.2.2019

Module Contents

Classes

CustomGraphicsScene	A custom QGraphicsScene. It provides signals to notify about items,
DesignGraphicsScene	A scene for the Design view.

class spinetoolbox.widgets.custom_qgraphicsscene.**CustomGraphicsScene**

Bases: PySide2.QtWidgets.QGraphicsScene

A custom QGraphicsScene. It provides signals to notify about items, and a method to center all items in the scene.

At the moment it's used by DesignGraphicsScene and the GraphViewMixin

item_move_finished

Emitted when an item has finished moving.

center_items()

Centers toplevel items in the scene.

class spinetoolbox.widgets.custom_qgraphicsscene.**DesignGraphicsScene**(*parent*, *toolbox*)

Bases: *CustomGraphicsScene*

A scene for the Design view.

Mainly, it handles drag and drop events of ProjectItemDragMixin sources.

Parameters

- **parent** (*QObject*) – scene's parent object
- **toolbox** (ToolboxUI) – reference to the main window

_handle_timeout()

clear_icons_and_links()

mouseMoveEvent(*event*)

Moves link drawer.

mousePressEvent(*event*)

Puts link drawer to sleep and log message if it looks like the user doesn't know what they're doing.

mouseReleaseEvent(*event*)

Makes link if drawer is released over a valid connector button.

_finish_link()

emit_connection_failed()

keyPressEvent(*event*)

Puts link drawer to sleep if user presses ESC.

connect_signals()

Connect scene signals.

project_item_icons()

handle_selection_changed()

Synchronizes selection with the project tree.

set_bg_color(*color*)

Change background color when this is changed in Settings.

Parameters
color (*QColor*) – Background color

set_bg_choice(*bg_choice*)

Set background choice when this is changed in Settings.

Parameters
bg (*str*) – "grid", "tree", or "solid"

dragLeaveEvent(*event*)

Accept event.

dragEnterEvent(*event*)

Accept event. Then call the super class method only if drag source is not a ProjectItemDragMixin.

dragMoveEvent(*event*)

Accept event. Then call the super class method only if drag source is not a ProjectItemDragMixin.

dropEvent(*event*)

Only accept drops when the source is an instance of ProjectItemDragMixin. Capture text from event's mimedata and show the appropriate 'Add Item form.'

event(*event*)

Accepts GraphicsSceneHelp events without doing anything, to not interfere with our usage of QToolTip.showText in graphics_items.ExclamationIcon.

drawBackground(*painter*, *rect*)

Reimplemented method to make a custom background.

Parameters

- **painter** (*QPainter*) – Painter that is used to paint background
- **rect** (*QRectF*) – The exposed (viewport) rectangle in scene coordinates

_draw_solid_bg(*painter*, *rect*)

Draws solid bg.

_draw_grid_bg(*painter*, *rect*)

Draws grid bg.

_draw_tree_bg(*painter*, *rect*)

Draws 'tree of life' bg.

select_link_drawer(*drawer_type*)

Selects current link drawer.

Parameters
drawer_type (LinkType) – selected link drawer's type

spinetoolbox.widgets.custom_qgraphicsviews

Classes for custom QGraphicsViews for the Design and Graph views.

authors

P. Savolainen (VTT), M. Marin (KTH)

date
6.2.2018

Module Contents

Classes

CustomQGraphicsView	Super class for Design and Entity QGraphicsViews.
DesignQGraphicsView	QGraphicsView for the Design View.

class spinetoolbox.widgets.custom_qgraphicsviews.**CustomQGraphicsView**(*parent*)

Bases: PySide2.QtWidgets.QGraphicsView

Super class for Design and Entity QGraphicsViews.

parent

Parent widget

Type
QWidget

Init CustomQGraphicsView.

abstract property _qsettings

property zoom_factor

reset_zoom()

Resets zoom to the default factor.

keyPressEvent(*event*)

Overridden method. Enable zooming with plus and minus keys (comma resets zoom). Send event downstream to QGraphicsItems if pressed key is not handled here.

Parameters
event (*QKeyEvent*) – Pressed key

mousePressEvent(*event*)

Set rubber band selection mode if Control pressed. Enable resetting the zoom factor from the middle mouse button.

mouseReleaseEvent(*event*)

Reestablish scroll hand drag mode.

_use_smooth_zoom()

wheelEvent(*event*)

Zooms in/out.

Parameters
event (*QWheelEvent*) – Mouse wheel event

resizeEvent(*event*)

Updates zoom if needed when the view is resized.

Parameters
event (*QResizeEvent*) – a resize event

setScene(*scene*)

Sets a new scene to this view.

Parameters
scene (*ShrinkingScene*) – a new scene

_handle_item_move_finished(*item*)

_update_zoom_limits()

Updates the minimum zoom limit and the zoom level with which the view fits all the items in the scene.

abstract _compute_max_zoom()

_handle_zoom_time_line_advanced(*pos*)

Performs zoom whenever the smooth zoom time line advances.

_handle_transformation_time_line_finished()

Cleans up after the smooth transformation time line finishes.

_handle_resize_time_line_finished()

Cleans up after resizing time line finishes.

zoom_in()

Perform a zoom in with a fixed scaling.

zoom_out()

Perform a zoom out with a fixed scaling.

gentle_zoom(*factor*, *zoom_focus=None*)

Perform a zoom by a given factor.

Parameters

- **factor** (*float*) – a scaling factor relative to the current scene scaling
- **zoom_focus** (*QPoint*) – focus of the zoom, e.g. mouse pointer position

_zoom(*factor*)

_get_viewport_scene_rect()

Returns the viewport rect mapped to the scene.

Returns
QRectF

_ensure_item_visible(*item*)

Resets zoom if item is not visible.

_set_preferred_scene_rect()

Sets the scene rect to the result of uniting the scene viewport rect and the items bounding rect.

class spinetoolbox.widgets.custom_qgraphicsviews.**DesignQGraphicsView**(*parent*)

Bases: *CustomQGraphicsView*

QGraphicsView for the Design View.

Parameters
parent (*QWidget*) – parent widget

property _qsettings

set_ui(*toolbox*)

Set a new scene into the Design View when app is started.

reset_zoom()

Resets zoom to the default factor.

_compute_max_zoom()

add_icon(*item_name*)

Adds project item's icon to the scene.

Parameters
item_name (*str*) – project item's name

remove_icon(*item_name*)

Removes project item's icon from scene.

Parameters
item_name (*str*) – name of the icon to remove

add_link(*src_connector*, *dst_connector*)

Pushes an AddLinkCommand to the toolbox undo stack.

Parameters

- **src_connector** (ConnectorButton) – source connector button
- **dst_connector** (ConnectorButton) – destination connector button

do_add_link(*connection*)

Adds given connection to the Design view.

Parameters
connection (*Connection*) – the connection to add

do_update_link(*updated_connection*)

Replaces a link on the Design view.

Parameters
updated_connection (*Connection*) – connection that was updated

remove_links(*links*)

Pushes a RemoveConnectionsCommand to the Toolbox undo stack.

Parameters
links (*list of* Link) – links to remove

do_remove_link(*connection*)

Removes a link from the scene.

Parameters
connection (*ConnectionBase*) – link's connection

remove_selected_links()

take_link(*link*)

Remove link, then start drawing another one from the same source connector.

add_jump(*src_connector*, *dst_connector*)

Pushes an AddJumpCommand to the Toolbox undo stack.

Parameters

- **src_connector** (ConnectorButton) – source connector button
- **dst_connector** (ConnectorButton) – destination connector button

do_add_jump(*jump*)

Adds given jump to the Design view.

Parameters
jump (*Jump*) – jump to add

do_update_jump(*updated_jump*)

Replaces a jump link on the Design view.

Parameters
updated_jump (*Jump*) – jump that was updated

do_remove_jump(*jump*)

Removes a jump from the scene.

Parameters
jump (*Jump*) – link's jump

contextMenuEvent(*event*)

Shows context menu for the blank view

Parameters
event (*QContextMenuEvent*) – Event

spinetoolbox.widgets.custom_qlineedits

Classes for custom line edits.

authors

M. Marin (KTH), P. Savolainen (VTT)

date
11.10.2018

Module Contents

Classes

PropertyQLineEdit	A custom QLineEdit for Project Item Properties.
CustomQLineEdit	A custom QLineEdit that accepts file drops and displays the path.

class spinetoolbox.widgets.custom_qlineedits.**PropertyQLineEdit**

Bases: *spinetoolbox.widgets.custom_qwidgets.UndoRedoMixin*, PySide2.QtWidgets.QLineEdit

A custom QLineEdit for Project Item Properties.

setText(*text*)

Overridden to prevent the cursor going to the end whenever the user is still editing. This happens because we set the text programmatically in undo/redo implementations.

class spinetoolbox.widgets.custom_qlineedits.**CustomQLineEdit**(**args*, ***kwargs*)

Bases: *spinetoolbox.widgets.custom_qwidgets.ElidedTextMixin*, *PropertyQLineEdit*

A custom QLineEdit that accepts file drops and displays the path.

parent

Parent for line edit widget

Type
QMainWindow

file_dropped

_set_full_text(*text*)

dragEnterEvent(*event*)

Accept a single file drop from the filesystem.

dragMoveEvent(*event*)

Accept event.

dropEvent(*event*)

Emit file_dropped signal with the file for the dropped url.

_elided_offset()

focusInEvent(*event*)

focusOutEvent(*event*)

_update_text(*text*)

spinetoolbox.widgets.custom_qtableview

Custom QTableView classes that support copy-paste and the like.

author

M. Marin (KTH)

date
18.5.2018

Module Contents

Classes

CopyPasteTableView	Custom QTableView class with copy and paste methods.
AutoFilterCopyPasteTableView	Custom QTableView class with autofilter functionality.
IndexedParameterValueTableViewBase	Custom QTableView base class with copy and paste methods for indexed parameter values.
TimeSeriesFixedResolutionTableView	A QTableView for fixed resolution time series table.
IndexedValueTableView	A QTableView class with for variable resolution time series and time patterns.
ArrayTableView	Custom QTableView with copy and paste methods for single column tables.
MapTableView	Custom QTableView with copy and paste methods for map tables.

Functions

_range(selection)	Returns the top left and bottom right corners of selection.
_could_be_time_stamp(s)	Evaluates if given string could be a time stamp.
system_lc_numeric()	

Attributes

_
_NOT_TIME_STAMP

`spinetoolbox.widgets.custom_qtableview._`

class `spinetoolbox.widgets.custom_qtableview.`**`CopyPasteTableView`**(*parent=None*)

Bases: `PySide2.QtWidgets.QTableView`

Custom QTableView class with copy and paste methods.

property copy_action

property paste_action

init_copy_and_paste_actions()

Initializes copy and paste actions and connects relevant signals.

set_external_copy_and_paste_actions(*copy_action*, *paste_action*)

Sets the view to use external copy and paste actions.

Note that this doesn't connect the actions' trigger signals; the owner of the actions is responsible for handling them.

Parameters

- **copy_action** (*QAction*) – copy action

- **paste_action** (*QAction*) – paste action

delete_content(*_=False*)

Deletes content from editable indexes in current selection.

can_copy()

copy(*_=False*)

Copies current selection to clipboard in excel format.

can_paste()

paste(*_=False*)

Paste data from clipboard.

static _read_pasted_text(*text*)

Parses a tab separated CSV text table.

Parameters
text (*str*) – a CSV formatted table

Returns
a list of rows

Return type
list

paste_on_selection()

Pastes clipboard data on selection, but not beyond. If data is smaller than selection, repeat data to fit selection.

paste_normal()

Pastes clipboard data, overwriting cells if needed.

class spinetoolbox.widgets.custom_qtableview.**AutoFilterCopyPasteTableView**(*parent*)

Bases: *CopyPasteTableView*

Custom QTableView class with autofilter functionality.

Parameters
parent (*QObject*) –

setModel(*model*)

Disconnects the sectionPressed signal which seems to be connected by the super method. Otherwise pressing the header just selects the column.

Parameters
model (*QAbstractItemModel*) –

_trigger_filter_menu(*_*)

Shows current column's auto filter menu.

show_auto_filter_menu(*logical_index*)

Called when user clicks on a horizontal section header. Shows/hides the auto filter widget.

Parameters
logical_index (*int*) – header section index

class spinetoolbox.widgets.custom_qtableview.**IndexedParameterValueTableViewBase**(*parent=None*)

Bases: *CopyPasteTableView*

Custom QTableView base class with copy and paste methods for indexed parameter values.

copy(*_=False*)

Copies current selection to clipboard in CSV format.

Returns
True if data was copied on the clipboard, False otherwise

Return type
bool

abstract static _read_pasted_text(*text*)

Reads CSV formatted table.

abstract paste(*_=False*)

Pastes data from clipboard to selection.

class spinetoolbox.widgets.custom_qtableview.**TimeSeriesFixedResolutionTableView**(*parent=None*)

Bases: *IndexedParameterValueTableViewBase*

A QTableView for fixed resolution time series table.

paste(*_=True*)

Pastes data from clipboard.

static _read_pasted_text(*text*)

Parses the given CSV table. Parsing is locale aware.

Parameters
text (*str*) – a CSV table containing numbers

Returns
A list of floats

Return type
list of float

_paste_to_values_column(*values*, *first_row*, *paste_length*)

Pastes data to the Values column.

Parameters

- **values** (*list*) – a list of float values to paste
- **first_row** (*int*) – index of the first row where to paste
- **paste_length** (*int*) – length of the paste selection (can be different from len(values))

Returns
A tuple (list(pasted indexes), list(pasted values))

Return type
tuple

class spinetoolbox.widgets.custom_qtableview.**IndexedValueTableView**(*parent=None*)

Bases: *IndexedParameterValueTableViewBase*

A QTableView class with for variable resolution time series and time patterns.

paste(*_=False*)

Pastes data from clipboard.

_paste_two_columns(*data_indexes*, *data_values*, *first_row*, *paste_length*)

Pastes data indexes and values.

Parameters

- **data_indexes** (*list*) – a list of data indexes (time stamps/durations)
- **data_values** (*list*) – a list of data values
- **first_row** (*int*) – first row index
- **paste_length** (*int*) – selection length for pasting

Returns
a tuple (modified model indexes, modified model values)

Return type
tuple

_paste_single_column(*values*, *first_row*, *first_column*, *paste_length*)

Pastes a single column of data.

Parameters

- **values** (*list*) – a list of data to paste (data indexes or values)
- **first_row** (*int*) – first row index
- **paste_length** (*int*) – selection length for pasting

Returns
a tuple (modified model indexes, modified model values)

Return type
tuple

static _read_pasted_text(*text*)

Parses a given CSV table.

Parameters
text (*str*) – a CSV table

Returns
a tuple (data indexes, data values)

Return type
tuple

class spinetoolbox.widgets.custom_qtableview.**ArrayTableView**(*parent=None*)

Bases: *IndexedParameterValueTableViewBase*

Custom QTableView with copy and paste methods for single column tables.

copy(*_=False*)

Copies current selection to clipboard in CSV format.

Returns
True if data was copied on the clipboard, False otherwise

Return type
bool

paste(_=*False*)

Pastes data from clipboard.

static _read_pasted_text(*text*)

Reads the first column of given CSV table.

Parameters
text (*str*) – a CSV table

Returns
data column

Return type
list of str

class spinetoolbox.widgets.custom_qtableview.**MapTableView**(*parent=None*)

Bases: *CopyPasteTableView*

Custom QTableView with copy and paste methods for map tables.

copy(_=*False*)

Copies current selection to clipboard in Excel compatible CSV format.

Returns
True if data was copied on the clipboard, False otherwise

Return type
bool

delete_content(_=*False*)

Deletes content in current selection.

paste(_=*False*)

Pastes data from clipboard.

Returns
True if data was pasted successfully, False otherwise

Return type
bool

static _read_pasted_text(*text*)

Parses a given CSV table.

Parameters
text (*str*) – a CSV table

Returns
a list of table rows

Return type
list of list

spinetoolbox.widgets.custom_qtableview.**_range**(*selection*)

Returns the top left and bottom right corners of selection.

Parameters
selection (*QItemSelection*) – a list of selected QItemSelection objects

Returns
a tuple (top row, bottom row, left column, right column)

Return type
tuple of ints

`spinetoolbox.widgets.custom_qtableview.`**`_NOT_TIME_STAMP`**

`spinetoolbox.widgets.custom_qtableview.`**`_could_be_time_stamp`**(*s*)

Evaluates if given string could be a time stamp.

This is to deal with special cases that are not intended as time stamps but could end up as one by the very greedy `DateTime` constructor.

Parameters
s (*str*) – string to evaluate

Returns
True if s could be a time stamp, False otherwise

Return type
bool

`spinetoolbox.widgets.custom_qtableview.`**`system_lc_numeric`**()

spinetoolbox.widgets.custom_qtextbrowser

Class for a custom QTextBrowser for showing the logs and tool output.

author

P. Savolainen (VTT)

date
6.2.2018

Module Contents

Classes

CustomQTextBrowser	Custom QTextBrowser class.
MonoSpaceFontTextBrowser	Custom QTextBrowser class.

class `spinetoolbox.widgets.custom_qtextbrowser.`**`CustomQTextBrowser`**(*parent*)

Bases: `PySide2.QtWidgets.QTextBrowser`

Custom QTextBrowser class.

Parameters
parent (*QWidget*) – Parent widget

`_ALL_RUNS = All executions`

`set_toolbox`(*toolbox*)

`append`(*text*)

Appends new text block to the end of the *original* document.

If the document contains more text blocks after the addition than a set limit, blocks are deleted at the start of the contents.

Parameters
text (*str*) – text to add

contextMenuEvent(*event*)

Reimplemented method to add a clear action into the default context menu.

Parameters
event (*QContextMenuEvent*) – Received event

clear()

_populate_executions_menu()

reset_executions_button_text()

_select_execution(*action*)

static _make_log_entry_title(*title*)

start_execution(*timestamp*)

Creates cursors (log entry points) for given items in event log.

Parameters
timestamp (*str*) – time stamp

add_log_message(*item_name*, *filter_id*, *message*)

Adds a message to an item's execution log.

Parameters

- **item_name** (*str*) – item name
- **filter_id** (*str*) – filter identifier
- **message** (*str*) – formatted message

execution_timestamps()

select_all_executions()

select_execution(*timestamp*)

_set_execution_visible(*timestamp*, *visible*)

set_item_log_selected(*selected*)

class spinetoolbox.widgets.custom_qtextbrowser.**MonoSpaceFontTextBrowser**(*parent*)

Bases: *CustomQTextBrowser*

Custom QTextBrowser class.

Parameters
parent (*QWidget*) – Parent widget

spinetoolbox.widgets.custom_qtreeview

Classes for custom QTreeView.

author

M. Marin (KTH)

date

25.4.2018

Module Contents

Classes

CopyTreeView	Custom QTreeView class with copy support.
SourcesTreeView	Custom QTreeView class for 'Sources' in Tool specification editor widget.
CustomTreeView	Custom QTreeView class for Tool specification editor form to enable keyPressEvent.

class spinetoolbox.widgets.custom_qtreeview.**CopyTreeView**(*parent*)

Bases: PySide2.QtWidgets.QTreeView

Custom QTreeView class with copy support.

Initialize the view.

can_copy()

Returns True if tree view has a selection to copy from.

Returns

True if there is something to copy

Return type

bool

static can_paste()

Returns False always as pasting is disabled into this view.

Returns

whether it is possible to paste to the view

Return type

bool

copy()

Copy current selection to clipboard in excel format.

class spinetoolbox.widgets.custom_qtreeview.**SourcesTreeView**(*parent*)

Bases: PySide2.QtWidgets.QTreeView

Custom QTreeView class for 'Sources' in Tool specification editor widget.

parent

The parent of this view

Type
QWidget

Initialize the view.

files_dropped

del_key_pressed

dragEnterEvent(*event*)

Accept file and folder drops from the filesystem.

dragMoveEvent(*event*)

Accept event.

dropEvent(*event*)

Emit files_dropped signal with a list of files for each dropped url.

keyPressEvent(*event*)

Overridden method to make the view support deleting items with a delete key.

class spinetoolbox.widgets.custom_qtreeview.**CustomTreeView**(*parent*)

Bases: PySide2.QtWidgets.QTreeView

Custom QTreeView class for Tool specification editor form to enable keyPressEvent.

parent

The parent of this view

Type
QWidget

Initialize the view.

del_key_pressed

keyPressEvent(*event*)

Overridden method to make the view support deleting items with a delete key.

spinetoolbox.widgets.custom_qwidgets

Custom QWidgets for Filtering and Zooming.

author

P. Vennström (VTT)

date
4.12.2018

Module Contents

Classes

ElidedTextMixin	
UndoRedoMixin	
FilterWidgetBase	Filter widget class.
CustomWidgetAction	A QWidgetAction with custom hovering.
ToolBarWidgetAction	An action with a tool bar.
ToolBarWidgetBase	A toolbar on the right, with enough space to print a text beneath.
ToolBarWidget	A toolbar on the right, with enough space to print a text beneath.
MenuItemToolBarWidget	A menu item with a toolbar on the right.
_MenuToolBar	A custom tool bar for `MenuItemToolBarWidget`.
TitleWidgetAction	A titled separator.
WrapLabel	A QLabel that always wraps text.
HyperTextLabel	A QLabel that supports hyperlinks.
QWizardProcessPage	A QWizards page with a log. Useful for pages that need to capture the output of a process.
LabelWithCopyButton	A read only QLabel with a QToolButton that copies the text to clipboard.
ElidedLabel	A QLabel with elided text.
HorizontalSpinBox	
PropertyQSpinBox	A spinbox where undo and redo key strokes apply to the project.
PurgeSettingsDialog	Purge settings dialog.

class spinetoolbox.widgets.custom_qwidgets.**ElidedTextMixin**(**args*, ***kwargs*)

setText(*text*)

_update_text(*text*)

_set_text_elided(*width=None*)

_elided_offset()

text()

resizeEvent(*event*)

class spinetoolbox.widgets.custom_qwidgets.**UndoRedoMixin**

keyPressEvent(*e*)

Overridden to catch and pass on the Undo and Redo commands when this line edit has the focus.

Parameters
e (*QKeyEvent*) – Event

class spinetoolbox.widgets.custom_qwidgets.**FilterWidgetBase**(*parent*)

Bases: PySide2.QtWidgets.QWidget

Filter widget class.

Init class.

Parameters
parent (*QWidget*) –

okPressed

cancelPressed

connect_signals()

save_state()

Saves the state of the FilterCheckboxListModel.

reset_state()

Sets the state of the FilterCheckboxListModel to saved state.

clear_filter()

Selects all items in FilterCheckBoxListModel.

has_filter()

Returns true if any item is filtered in FilterCheckboxListModel false otherwise.

set_filter_list(*data*)

Sets the list of items to filter.

_apply_filter()

Apply current filter and save state.

_cancel_filter()

Cancel current edit of filter and set the state to the stored state.

_filter_list()

Filter list with current text.

_text_edited(*new_text*)

Callback for edit text, starts/restarts timer. Start timer after text is edited, restart timer if text is edited before last time out.

class spinetoolbox.widgets.custom_qwidgets.**CustomWidgetAction**(*parent=None*)

Bases: PySide2.QtWidgets.QWidgetAction

A QWidgetAction with custom hovering.

Class constructor.

Parameters
parent (*QMenu*) – the widget's parent

_handle_hovered()

Hides other menus that might be shown in the parent widget and repaints it. This is to emulate the behavior of QAction.

class spinetoolbox.widgets.custom_qwidgets.**ToolBarWidgetAction**(*text*, *parent=None*, *compact=False*)

Bases: *CustomWidgetAction*

An action with a tool bar.

tool_bar

Type
QToolBar

Class constructor.

Parameters
parent (*QMenu*) – the widget's parent

eventFilter(*obj*, *ev*)

_handle_hovered()

Hides other menus that might be shown in the parent widget and repaints it. This is to emulate the behavior of QAction.

class spinetoolbox.widgets.custom_qwidgets.**ToolBarWidgetBase**(*text*, *parent=None*)

Bases: PySide2.QtWidgets.QWidget

A toolbar on the right, with enough space to print a text beneath.

tool_bar

Type
QToolBar

Class constructor.

Parameters

- **text** (*str*) –
- **parent** (*QWidget*) – the widget's parent

class spinetoolbox.widgets.custom_qwidgets.**ToolBarWidget**(*text*, *parent=None*)

Bases: *ToolBarWidgetBase*

A toolbar on the right, with enough space to print a text beneath.

tool_bar

Type
QToolBar

Class constructor.

Parameters

- **text** (*str*) –
- **parent** (*QWidget*) – the widget's parent

class spinetoolbox.widgets.custom_qwidgets.**MenuItemToolBarWidget**(*text*, *parent=None*, *compact=False*)

Bases: *ToolBarWidgetBase*

A menu item with a toolbar on the right.

tool_bar

Type
QToolBar

Class constructor.

Parameters

- **text** (*str*) –
- **parent** (*QWidget*) – the widget's parent
- **compact** (*bool*) – if True, the widget uses the minimal space

paintEvent(*event*)

Draws the menu item, then calls the super() method to draw the tool bar.

class spinetoolbox.widgets.custom_qwidgets.**_MenuToolBar**(*parent=None*)

Bases: PySide2.QtWidgets.QToolBar

A custom tool bar for MenuItemToolBarWidget.

enabled_changed

_align_buttons()

Align all buttons to bottom so frames look good.

add_frame(*left*, *right*, *title*)

Add frame around given actions, with given title.

Parameters

- **left** (*QAction*) –
- **right** (*QAction*) –
- **title** (*str*) –

is_enabled()

addActions(*actions*)

Overriden method to customize tool buttons.

addAction(**args*, ***kwargs*)

Overriden method to customize the tool button.

sizeHint()

Make room for frames if needed.

paintEvent(*ev*)

Paint the frames.

_setup_action_button(*action*)

Customizes the QToolButton associated with given action:

1. Makes sure that the text honors the action's mnemonics.
2. Installs this as event filter on the button (see self.eventFilter()).

Must be called everytime an action is added to the tool bar.

Parameters
QAction –

actionEvent(*ev*)

Updates `self._enabled`: True if at least one non-separator action is enabled, False otherwise. Emits `self.enabled_changed` accordingly.

eventFilter(*obj*, *ev*)

Installed on each action's QToolButton. Ignores Up and Down key press events, so they are handled by the toolbar for custom navigation.

keyPressEvent(*ev*)

Navigates over the tool bar buttons.

hideEvent(*ev*)

class spinetoolbox.widgets.custom_qwidgets.**TitleWidgetAction**(*title*, *parent=None*)

Bases: *CustomWidgetAction*

A titled separator.

Class constructor.

Parameters
parent (*QMenu*) – the widget's parent

H_MARGIN = 5

V_MARGIN = 2

static _add_line(*widget*, *layout*)

isSeparator()

class spinetoolbox.widgets.custom_qwidgets.**WrapLabel**(*text=''*, *parent=None*)

Bases: `PySide2.QtWidgets.QLabel`

A QLabel that always wraps text.

class spinetoolbox.widgets.custom_qwidgets.**HyperTextLabel**(*text=''*, *parent=None*)

Bases: *WrapLabel*

A QLabel that supports hyperlinks.

class spinetoolbox.widgets.custom_qwidgets.**QWizardProcessPage**(*parent*)

Bases: `PySide2.QtWidgets.QWizardPage`

A QWizards page with a log. Useful for pages that need to capture the output of a process.

class _ExecutionManager

A descriptor that stores a QProcessExecutionManager. When `execution_finished` is emitted, it shows the button to copy the process log.

public_name

private_name

__set_name__(*owner*, *name*)

__get__(*obj*, *objtype=None*)

__set__(*obj*, *value*)

msg

msg_warning

msg_error

msg_success

msg_proc

msg_proc_error

_exec_mngr

_connect_signals()

_handle_copy_clicked(*_=False*)

_add_msg(*msg*)

_add_msg_warning(*msg*)

_add_msg_error(*msg*)

_add_msg_succes(*msg*)

isComplete()

cleanupPage()

class spinetoolbox.widgets.custom_qwidgets.**LabelWithCopyButton**(*text=''*, *parent=None*)

Bases: PySide2.QtWidgets.QWidget

A read only QLabel with a QToolButton that copies the text to clipboard.

class spinetoolbox.widgets.custom_qwidgets.**ElidedLabel**(**args*, ***kwargs*)

Bases: *ElidedTextMixin*, PySide2.QtWidgets.QLabel

A QLabel with elided text.

class spinetoolbox.widgets.custom_qwidgets.**HorizontalSpinBox**(**args*, ***kwargs*)

Bases: PySide2.QtWidgets.QToolBar

valueChanged

value()

setMinimum(*minimum*)

setValue(*value*, *strict=False*)

_dec_value()

_inc_value()

_focus_line_edit()

class spinetoolbox.widgets.custom_qwidgets.**PropertyQSpinBox**

Bases: *UndoRedoMixin*, PySide2.QtWidgets.QSpinBox

A spinbox where undo and redo key strokes apply to the project.

class spinetoolbox.widgets.custom_qwidgets.**PurgeSettingsDialog**(*purge_settings*, *parent=None*)

Bases: PySide2.QtWidgets.QDialog

Purge settings dialog.

Parameters

- **purge_settings** (*dict*) – purge settings
- **parent** (*QWidget, optional*) – parent widget

get_purge_settings()

Returns current purge settings.

Returns
mapping from purgeable database item name to purge flag

Return type
dict

spinetoolbox.widgets.datetime_editor

An editor widget for editing datetime database (relationship) parameter values.

author

A. Soininen (VTT)

date
28.6.2019

Module Contents

Classes

DatetimeEditor	An editor widget for DateTime type parameter values.

Functions

_QDateTime_to_datetime(dt)	Converts a QDateTime object to Python's datetime.datetime type.
_datetime_to_QDateTime(dt)	Converts Python's datetime.datetime object to QDateTime.

spinetoolbox.widgets.datetime_editor.**_QDateTime_to_datetime**(*dt*)

Converts a QDateTime object to Python's datetime.datetime type.

spinetoolbox.widgets.datetime_editor.**_datetime_to_QDateTime**(*dt*)

Converts Python's datetime.datetime object to QDateTime.

class spinetoolbox.widgets.datetime_editor.**DatetimeEditor**(*parent=None*)

Bases: PySide2.QtWidgets.QWidget

An editor widget for DateTime type parameter values.

parent

a parent widget

Type
QWidget

_change_datetime(*new_datetime*)

Updates the internal DateTime value

set_value(*value*)

Sets the value to be edited.

value()

Returns the editor's current value.

spinetoolbox.widgets.duration_editor

An editor widget for editing duration database (relationship) parameter values.

author

A. Soininen (VTT)

date
28.6.2019

Module Contents

Classes

DurationEditor	An editor widget for Duration type parameter values.

class spinetoolbox.widgets.duration_editor.**DurationEditor**(*parent=None*)

Bases: PySide2.QtWidgets.QWidget

An editor widget for Duration type parameter values.

parent

a parent widget

Type
QWidget

_change_duration()

Updates the value being edited.

set_value(*value*)

Sets the value for editing.

value()

Returns the current Duration.

spinetoolbox.widgets.indexed_value_table_context_menu

Context menus for parameter value editor widgets.

author

A. Soininen (VTT)

date

5.7.2019

Module Contents

Classes

ContextMenuBase	Context menu base for parameter value editor tables.
ArrayTableContextMenu	Context menu for array editor tables.
IndexedValueTableContextMenu	Context menu for time series and time pattern editor tables.
MapTableContextMenu	Context menu for map editor tables.

Functions

_unique_row_ranges(selections)	Merged ranges in given selections to unique ranges.
_unique_column_ranges(selections)	Merged ranges in given selections to unique ranges.
_merge_intervals(intervals)	Merges given intervals if they overlap.

Attributes

_INSERT_SINGLE_COLUMN_AFTER
_INSERT_SINGLE_ROW_AFTER
_INSERT_MULTIPLE_COLUMNS_AFTER
_INSERT_MULTIPLE_ROWS_AFTER
_INSERT_SINGLE_COLUMN_BEFORE
_INSERT_SINGLE_ROW_BEFORE
_INSERT_MULTIPLE_COLUMNS_BEFORE
_INSERT_MULTIPLE_ROWS_BEFORE
_OPEN_EDITOR
_PLOT
_PLOT_IN_WINDOW
_REMOVE_COLUMNS
_REMOVE_ROWS
_TRIM_COLUMNS

spinetoolbox.widgets.indexed_value_table_context_menu.**_INSERT_SINGLE_COLUMN_AFTER** = **Insert column after**

spinetoolbox.widgets.indexed_value_table_context_menu.**_INSERT_SINGLE_ROW_AFTER** = **Insert row after**

spinetoolbox.widgets.indexed_value_table_context_menu.**_INSERT_MULTIPLE_COLUMNS_AFTER** = **Insert columns after...**

spinetoolbox.widgets.indexed_value_table_context_menu.**_INSERT_MULTIPLE_ROWS_AFTER** = **Insert rows after...**

spinetoolbox.widgets.indexed_value_table_context_menu.**_INSERT_SINGLE_COLUMN_BEFORE** = **Insert column before**

spinetoolbox.widgets.indexed_value_table_context_menu.**_INSERT_SINGLE_ROW_BEFORE** = **Insert row before**

spinetoolbox.widgets.indexed_value_table_context_menu.**_INSERT_MULTIPLE_COLUMNS_BEFORE** = **Insert columns before...**

spinetoolbox.widgets.indexed_value_table_context_menu.**_INSERT_MULTIPLE_ROWS_BEFORE = Insert rows before...**

spinetoolbox.widgets.indexed_value_table_context_menu.**_OPEN_EDITOR = Edit...**

spinetoolbox.widgets.indexed_value_table_context_menu.**_PLOT = Plot...**

spinetoolbox.widgets.indexed_value_table_context_menu.**_PLOT_IN_WINDOW = Plot in window**

spinetoolbox.widgets.indexed_value_table_context_menu.**_REMOVE_COLUMNS = Remove columns**

spinetoolbox.widgets.indexed_value_table_context_menu.**_REMOVE_ROWS = Remove rows**

spinetoolbox.widgets.indexed_value_table_context_menu.**_TRIM_COLUMNS = Trim columns**

class spinetoolbox.widgets.indexed_value_table_context_menu.**ContextMenuBase**(*table_view*, *position*)

Bases: PySide2.QtWidgets.QMenu

Context menu base for parameter value editor tables.

Parameters

- **table_view** (*QTableView*) – the view where the menu is invoked
- **position** (*QPoint*) – menu's position on the table view

_add_default_actions()

Adds default actions to the menu.

_first_row()

Returns the first selected row.

Returns
index to the first row

Return type
int

_insert_multiple_rows_after()

Prompts for row count, then inserts new rows below the current selection.

_insert_multiple_rows_before()

Prompts for row count, then inserts new rows above the current selection.

_insert_single_row_after()

Inserts a single row below the current selection.

_insert_single_row_before()

Inserts a single row above the current selection.

_last_row()

Returns the last selected row.

Returns
index to the last row

Return type
int

_prompt_row_count()

Prompts for number of rows to insert.

Returns
number of rows

Return type
int

_remove_rows()

Removes selected rows.

class spinetoolbox.widgets.indexed_value_table_context_menu.**ArrayTableContextMenu**(*editor*, *table_view*, *position*)

Bases: *ContextMenuBase*

Context menu for array editor tables.

Parameters

- **editor** (*ArrayEditor*) – array editor widget
- **table_view** (*QTableView*) – the view where the menu is invoked
- **position** (*QPoint*) – menu's position

_show_value_editor()

Opens the value element editor.

class spinetoolbox.widgets.indexed_value_table_context_menu.**IndexedValueTableContextMenu**(*table_view*, *position*)

Bases: *ContextMenuBase*

Context menu for time series and time pattern editor tables.

Parameters

- **table_view** (*QTableView*) – the view where the menu is invoked
- **position** (*QPoint*) – menu's position

class spinetoolbox.widgets.indexed_value_table_context_menu.**MapTableContextMenu**(*editor*, *table_view*, *position*)

Bases: *ContextMenuBase*

Context menu for map editor tables.

Parameters

- **editor** (*MapEditor*) – map editor widget
- **table_view** (*QTableView*) – the view where the menu is invoked
- **position** (*QPoint*) – table cell index

_first_column()

Returns the first selected column.

Returns
index to the first column

Return type
int

_insert_multiple_columns_after()

Prompts for column count, then inserts new columns right from the current selection.

_insert_multiple_columns_before()

Prompts for column count, then inserts new columns left from the current selection.

_insert_single_column_before()

Inserts a single column left from the current selection.

_insert_single_column_after()

Inserts a single column right from the current selection.

_last_column()

Returns the last selected column.

Returns
index to the last column

Return type
int

_prompt_column_count()

Prompts for number of column to insert.

Returns
number of columns

Return type
int

_remove_columns()

Removes selected columns

_show_value_editor()

Opens the value element editor.

_plot(*checked=False*)

Plots current indexes.

_plot_in_window(*action*)

Plots the selected cells in an existing window.

_trim_columns()

Removes excessive columns from the table.

spinetoolbox.widgets.indexed_value_table_context_menu.**_unique_row_ranges**(*selections*)

Merged ranges in given selections to unique ranges.

Parameters
selections (*list of QItemSelectionRange*) – selected ranges

Returns
a list of [first_row, last_row] ranges

Return type
list of list

`spinetoolbox.widgets.indexed_value_table_context_menu.`**`_unique_column_ranges`**(*selections*)

Merged ranges in given selections to unique ranges.

Parameters
selections (*list of QItemSelectionRange*) – selected ranges

Returns
a list of [first_row, last_row] ranges

Return type
list of list

`spinetoolbox.widgets.indexed_value_table_context_menu.`**`_merge_intervals`**(*intervals*)

Merges given intervals if they overlap.

Parameters
intervals (*list of list*) – a list of intervals in the form [first, last]

Returns
merged intervals in the form [first, last]

Return type
list of list

spinetoolbox.widgets.install_julia_wizard

Classes for custom QDialogs for julia setup.

author

M. Marin (KTH)

date
13.5.2018

Module Contents

Classes

_PageId	Enum where members are also (and must be) ints
InstallJuliaWizard	A wizard to install julia
JillNotFoundPage	
IntroPage	
SelectDirsPage	
InstallJuliaPage	A QWizards page with a log. Useful for pages that need to capture the output of a process.
SuccessPage	
FailurePage	

Attributes

jill_install

spinetoolbox.widgets.install_julia_wizard.**jill_install**

class spinetoolbox.widgets.install_julia_wizard.**_PageId**

Bases: enum.IntEnum

Enum where members are also (and must be) ints

Initialize self. See help(type(self)) for accurate signature.

INTRO

SELECT_DIRS

INSTALL

SUCCESS

FAILURE

class spinetoolbox.widgets.install_julia_wizard.**InstallJuliaWizard**(*parent*)

Bases: PySide2.QtWidgets.QWizard

A wizard to install julia

Initialize class.

Parameters
parent (*QWidget*) – the parent widget (SettingsWidget)

julia_exe_selected

set_julia_exe()

accept()

class spinetoolbox.widgets.install_julia_wizard.**JillNotFoundPage**(*parent*)

Bases: PySide2.QtWidgets.QWizardPage

class spinetoolbox.widgets.install_julia_wizard.**IntroPage**(*parent*)

Bases: PySide2.QtWidgets.QWizardPage

nextId()

class spinetoolbox.widgets.install_julia_wizard.**SelectDirsPage**(*parent*)

Bases: PySide2.QtWidgets.QWizardPage

initializePage()

_select_install_dir()

_select_symlink_dir()

nextId()

class spinetoolbox.widgets.install_julia_wizard.**InstallJuliaPage**(*parent*)

Bases: *spinetoolbox.widgets.custom_qwidgets.QWizardProcessPage*

A QWizards page with a log. Useful for pages that need to capture the output of a process.

cleanupPage()

initializePage()

_handle_julia_install_finished(*ret*)

nextId()

class spinetoolbox.widgets.install_julia_wizard.**SuccessPage**(*parent*)

Bases: PySide2.QtWidgets.QWizardPage

initializePage()

nextId()

class spinetoolbox.widgets.install_julia_wizard.**FailurePage**(*parent*)

Bases: PySide2.QtWidgets.QWizardPage

initializePage()

nextId()

spinetoolbox.widgets.jump_properties_widget

Contains jump properties widget's business logic.

author

A. Soininen (VTT)

date

23.6.2021

Module Contents

Classes

JumpPropertiesWidget	Widget for jump link properties.

class spinetoolbox.widgets.jump_properties_widget.**JumpPropertiesWidget**(*toolbox*, *base_color=None*)

Bases: *spinetoolbox.widgets.properties_widget.PropertiesWidgetBase*

Widget for jump link properties.

Parameters

toolbox (ToolboxUI) – The toolbox instance where this widget should be embedded

_load_condition_into_ui(*condition*)

_make_condition_from_ui()

_change_condition()

Stores jump condition to link.

_show_tool_spec_form(*_checked=False*)

_set_save_script_button_enabled()

_update_add_args_button_enabled(*_selected*, *_deselected*)

_do_update_add_args_button_enabled()

_update_remove_args_button_enabled(*_selected*, *_deselected*)

_do_update_remove_args_button_enabled()

_populate_cmd_line_args_model()

_push_update_cmd_line_args_command(*cmd_line_args*)

_remove_arg(*_=False*)

_add_args(*_=False*)

set_link(*jump*)

Hooks the widget to given jump link, so that user actions are reflected in the jump link's configuration.

Parameters

jump (LoggingJump) – link to hook into

unset_link()

Releases the widget from any links.

set_condition(*jump*, *condition*)

update_cmd_line_args(*jump*, *cmd_line_args*)

spinetoolbox.widgets.jupyter_console_widget

Class for a custom RichJupyterWidget that can run Tool instances.

authors

M. Marin (KTH), P. Savolainen (VTT)

date

22.10.2019

Module Contents

Classes

JupyterConsoleWidget	Base class for all embedded console widgets that can run tool instances.

Attributes

traitlets_logger
asyncio_logger

spinetoolbox.widgets.jupyter_console_widget.**traitlets_logger**

spinetoolbox.widgets.jupyter_console_widget.**asyncio_logger**

class spinetoolbox.widgets.jupyter_console_widget.**JupyterConsoleWidget**(*toolbox*, *target_kernel_name*, *owner=None*)

Bases: `qtconsole.rich_jupyter_widget.RichJupyterWidget`

Base class for all embedded console widgets that can run tool instances.

Parameters

- **toolbox** (`ToolboxUI`) – QMainWindow instance
- **target_kernel_name** (*str*) – Kernel name, e.g. 'julia-1.6'
- **owner** (`ProjectItem`, *NoneType*) – Item that owns the console.

property owner_names

name()

Returns console name for display purposes.

start_console(*_=False*)

Starts chosen Python/Julia kernel if available and not already running. Context menu start action handler.

restart_console(*_=False*)

Restarts current Python/Julia kernel. Starts a new kernel if it is not running or if chosen kernel has been changed in Settings. Context menu restart action handler.

call_start_kernel()

Finds a valid kernel and calls `start_kernel()` with it.

start_kernel(*k_path*)

Starts a kernel manager and kernel client and attaches the client to this Console.

Parameters
k_path (*str*) – Directory where the the kernel specs are located

shutdown_kernel()

Shut down Julia/Python kernel.

dragEnterEvent(*e*)

Don't accept project item drops.

_handle_status(*msg*)

Handles status message.

enterEvent(*event*)

Sets busy cursor during console (re)starts.

abstract _is_complete(*source*, *interactive*)

See base class.

_context_menu_make(*pos*)

Reimplemented to add actions to console context-menus.

copy_input()

Copies only input.

_replace_client()

connect_to_kernel(*kernel_name*, *connection_file*)

Connects to an existing kernel. Used when Spine Engine is managing the kernel for project execution.

Parameters

- **kernel_name** (*str*) –
- **connection_file** (*str*) – Path to the connection file of the kernel

interrupt()

[TODO: Remove?] Sends interrupt signal to kernel.

spinetoolbox.widgets.kernel_editor

Dialog for selecting a kernel or creating a new Julia or Python kernel.

author

P. Savolainen (VTT)

date

7.10.2020

Module Contents

Classes

KernelEditorBase	Base class for kernel editors.
KernelEditor	Class for a Python and Julia kernel editor.
MiniKernelEditorBase	Base class for kernel editors.
MiniPythonKernelEditor	A reduced version of KernelEditor that basically just takes care of installing one Python kernel.
MiniJuliaKernelEditor	A reduced version of KernelEditor that basically just takes care of installing one Julia kernel.

Functions

find_kernels()	Returns a dictionary mapping kernel names to kernel paths.
find_python_kernels()	Returns a dictionary of Python kernels. Keys are kernel_names, values are kernel paths.
find_julia_kernels()	Returns a dictionary of Julia kernels. Keys are kernel_names, values are kernel paths.
find_unknown_kernels()	Returns a dictionary of kernels that are neither Python nor Julia kernels.
format_event_message(msg_type, message[, show_datetime])	Formats message for the kernel editor text browser.
format_process_message(msg_type, message)	Formats process message for the kernel editor text browser.

class spinetoolbox.widgets.kernel_editor.**KernelEditorBase**(*parent*, *python_or_julia*)

Bases: PySide2.QtWidgets.QDialog

Base class for kernel editors.

Parameters

- **parent** (*QSettingsWidget*) – Toolbox settings widget
- **python_or_julia** (*str*) – kernel type; valid values: "julia", "python"

setup_dialog_style()

Sets windows icon and stylesheet. This can be removed when SettingsWidget inherits stylesheet from ToolboxUI.

connect_signals()

Connects signals to slots.

check_options(*prgm*, *kernel_name*, *display_name*, *python_or_julia*)

Checks that user options are valid before advancing with kernel making.

Parameters

- **prgm** (*str*) – Full path to Python or Julia program
- **kernel_name** (*str*) – Kernel name
- **display_name** (*str*) – Kernel display name
- **python_or_julia** (*str*) – Either 'python' or 'julia'

Returns

True if all user input is valid for making a new kernel, False otherwise

Return type

bool

abstract _python_kernel_name()

abstract _python_kernel_display_name()

abstract _python_interpreter_name()

make_python_kernel(*checked=False*)

Makes a new Python kernel. Offers to install ipykernel package if it is missing from the selected Python environment. Overwrites existing kernel with the same name if this is ok by user.

static is_package_installed(*python_path*, *package_name*)

Checks if given package is installed to given Python environment.

Parameters

- **python_path** (*str*) – Full path to selected Python interpreter
- **package_name** (*str*) – Package name

Returns
True if installed, False if not

Return type
(bool)

start_package_install_process(*python_path*, *package_name*)

Starts installing the given package using pip.

Parameters

- **python_path** (*str*) – Full path to selected Python interpreter
- **package_name** (*str*) – Package name to install using pip

handle_package_install_process_finished(*retval*)

Handles installing package finished.

Parameters
retval (*int*) – Process return value. 0: success, !0: failure

start_kernelspec_install_process(*prgm*, *k_name*, *d_name*)

Installs kernel specifications for the given Python environment. Runs e.g. this command in QProcess

python -m ipykernel install –user –name python-X.Y –display-name PythonX.Y

Creates new kernel specs into %APPDATA%jupyterkernels. Existing directory will be overwritten.

Note: We cannot use –sys.prefix here because if we have selected to create a kernel for some other python that was used in launching the app, the kernel will be created into a location that is not discoverable by jupyter and hence not by Spine Toolbox. E.g. when sys.executable is C:Python36python.exe, and we have selected that as the python for Spine Toolbox (Settings->Tools->Python interpreter is empty), creating a kernel with –sys-prefix creates kernel specs into C:Python36sharejupyterkernelspython-3.6. This is ok and the kernel spec is discoverable by jupyter and Spine Toolbox.

BUT when sys.executable is C:Python36python.exe, and we have selected another python for Spine Toolbox (Settings->Tools->Python interpreter is C:Python38python.exe), creating a kernel with –sys-prefix creates a kernel into C:Python38sharejupyterkernelspython-3.8-sys-prefix. This is not discoverable by jupyter nor Spine Toolbox. You would need to start the app using C:Python38python.exe to see and use that kernel spec.

Using –user option instead, creates kernel specs that are discoverable by any python that was used in starting Spine Toolbox.

Parameters

- **prgm** (*str*) – Full path to Python interpreter for which the kernel is created
- **k_name** (*str*) – Kernel name
- **d_name** (*str*) – Kernel display name

handle_kernelspec_install_process_finished(*retval*)

Handles case when the process for installing the kernel has finished.

Parameters

retval (*int*) – Process return value. 0: success, !0: failure

abstract _julia_kernel_name()

abstract _julia_executable()

abstract _julia_project()

make_julia_kernel(*checked=False*)

Makes a new Julia kernel. Offers to install IJulia package if it is missing from the selected Julia project. Overwrites existing kernel with the same name if this is ok by user.

_is_rebuild_ijulia_needed()

is_ijulia_installed(*program*, *project*)

Checks if IJulia is installed for the given project. Note: Trying command 'using IJulia' does not work since it automatically tries loading it from the LOAD_PATH if not it's not found in the active project.

Returns

0 when process failed to start, 1 when IJulia is installed, 2 when IJulia is not installed.

Return type

int

start_ijulia_install_process(*julia*, *project*)

Starts installing IJulia package to given Julia project.

Parameters

- **julia** (*str*) – Full path to selected Julia executable
- **project** (*str*) – Julia project (e.g. dir path or '@.', or '.')

handle_ijulia_install_finished(*ret*)

Runs when IJulia install process finishes.

Parameters

ret (*int*) – Process return value. 0: success, !0: failure

start_ijulia_rebuild_process(*program*, *project*)

Starts rebuilding IJulia.

handle_ijulia_rebuild_finished(*ret*)

Runs when IJulia rebuild process finishes.

Parameters

ret (*int*) – Process return value. 0: success, !0: failure

start_ijulia_installkernel_process(*program*, *project*, *kernel_name*)

Installs the kernel using IJulia.installkernel function. Given kernel_name is actually the new kernel DISPLAY name. IJulia strips the whitespace and uncapitalizes this to make the kernel name automatically. Julia version is concatenated to both names automatically (This cannot be changed).

handle_installkernel_process_finished(*retval*)

Checks whether or not the IJulia.installkernel process finished successfully.

Parameters

retval (*int*) – Process return value. 0: success, !0: failure

restore_dialog_dimensions()

Restore widget location, dimensions, and state from previous session.

add_message(*msg*)

Append regular message to kernel editor text browser.

Parameters

msg (*str*) – String written to QTextBrowser

add_success_message(*msg*)

Append message with green text color to kernel editor text browser.

Parameters

msg (*str*) – String written to QTextBrowser

add_error_message(*msg*)

Append message with red color to kernel editor text browser.

Parameters

msg (*str*) – String written to QTextBrowser

add_warning_message(*msg*)

Append message with yellow (golden) color to kernel editor text browser.

Parameters

msg (*str*) – String written to QTextBrowser

add_process_message(*msg*)

Writes message from stdout to kernel editor text browser.

Parameters

msg (*str*) – String written to QTextBrowser

add_process_error_message(*msg*)

Writes message from stderr to kernel editor text browser.

Parameters

msg (*str*) – String written to QTextBrowser

_save_ui()

class spinetoolbox.widgets.kernel_editor.**KernelEditor**(*parent*, *python*, *julia*, *python_or_julia*, *current_kernel*)

Bases: *KernelEditorBase*

Class for a Python and Julia kernel editor.

Parameters

- **parent** (*QWidget*) – Parent widget (Settings widget)
- **python** (*str*) – Python interpreter, may be empty string
- **julia** (*str*) – Julia executable, may be empty string
- **python_or_julia** (*str*) – Setup KernelEditor according to selected mode
- **current_kernel** (*str*) – Current selected Python or Julia kernel name

connect_signals()

Connects signals to slots.

_julia_kernel_name()

_julia_executable()

_julia_project()

_python_kernel_name()

_python_kernel_display_name()

_python_interpreter_name()

_handle_kernel_selection_changed(*_selected*, *_deselected*)

_update_ok_button_enabled()

python_kernel_name_edited(*txt*)

Updates the display name place holder text and the command QCustomLabel tool tip.

select_julia_clicked(*checked=False*)

Opens file browser where user can select a Julia executable for the new kernel.

select_julia_project_clicked(*checked=False*)

Opens file browser where user can select a Julia project path for the new kernel.

select_python_clicked(*checked=False*)

Opens file browser where user can select the python interpreter for the new kernel.

update_python_cmd_tooltip()

Updates Python command (CustomQLabel) tooltip according to selections.

update_julia_cmd_tooltip()

Updates Julia command (CustomQLabel) tooltip according to selections.

set_kernel_selected(*k_name*)

Finds row index of given kernel name from the model, sets it selected and scrolls the view so that it's visible.

Parameters
k_name (*str*) – Kernel name to find and select

_check_kernel_is_ok(*current*, *previous*)

Shows a notification if there are any known problems with selected kernel.

Parameters

- **current** (*QModelIndex*) – Currently selected index
- **previous** (*QModelIndex*) – Previously selected index

find_column(*label*)

Returns the column number from the kernel model with the given label.

Parameters
label (*str*) – Header column label

Returns
Column number or -1 if label not found

Return type
int

check_options(*prgm*, *kernel_name*, *display_name*, *python_or_julia*)

Checks that user options are valid before advancing with kernel making.

Parameters

- **prgm** (*str*) – Full path to Python or Julia program
- **kernel_name** (*str*) – Kernel name
- **display_name** (*str*) – Kernel display name
- **python_or_julia** (*str*) – Either 'python' or 'julia'

Returns
True if all user input is valid for making a new kernel, False otherwise

Return type
bool

_is_rebuild_ijulia_needed()

handle_kernelspec_install_process_finished(*retval*)

Handles case when the process for installing the kernel has finished.

Parameters
retval (*int*) – Process return value. 0: success, !0: failure

handle_installkernel_process_finished(*retval*)

Checks whether or not the IJulia.installkernel process finished successfully.

Parameters
retval (*int*) – Process return value. 0: success, !0: failure

populate_kernel_model()

Populates the kernel model with kernels found in user's system either with Python or Julia kernels. Unknows, invalid, and unsupported kernels are appended to the end.

static get_kernel_deats(*kernel_path*)

Reads kernel.json from given kernel path and returns the details in a dictionary.

Parameters
kernel_path (*str*) – Full path to kernel directory

Returns
language (str), path to interpreter (str), display name (str), project (str) (NA for Python kernels)

Return type
dict

show_kernel_list_context_menu(*pos*)

Shows the context-menu in the kernel list table view.

_open_kernel_json(*checked=False*)

Opens kernel.json file using the default application for .json files.

_open_kernel_dir(*checked=False*)

Opens kernel directory in OS file browser.

_remove_kernel(*checked=False*)

Removes selected kernel by deleting the kernel directory.

mousePressEvent(*e*)

Saves mouse position at the start of dragging.

Parameters
e (*QMouseEvent*) – Mouse event

mouseReleaseEvent(*e*)

Saves mouse position at the end of dragging.

Parameters
e (*QMouseEvent*) – Mouse event

mouseMoveEvent(*e*)

Moves the window when mouse button is pressed and mouse cursor is moved.

Parameters
e (*QMouseEvent*) – Mouse event

done(*r*)

Overridden QDialog method. Sets the selected kernel instance attribute so that it can be read by the SettingsForm after this dialog has been closed.

Parameters
r (*int*) –

closeEvent(*event=None*)

Handles dialog closing.

Parameters
event (*QCloseEvent*) – Close event

class spinetoolbox.widgets.kernel_editor.**MiniKernelEditorBase**(*parent*, *python_or_julia*)

Bases: *KernelEditorBase*

Base class for kernel editors.

Parameters

- **parent** (*QSettingsWidget*) – Toolbox settings widget
- **python_or_julia** (*str*) – kernel type; valid values: "julia", "python"

_python_interpreter_name()

_julia_executable()

_julia_project()

_show_close_button(*failed=False*)

make_kernel()

abstract _do_make_kernel()

class spinetoolbox.widgets.kernel_editor.**MiniPythonKernelEditor**(*parent*, *python_exe*)

Bases: *MiniKernelEditorBase*

A reduced version of KernelEditor that basically just takes care of installing one Python kernel. The python exe is passed in the constructor, then calling `make_kernel` starts the process.

Parameters

- **parent** (*QSettingsWidget*) – Toolbox settings widget

- **python_or_julia** (*str*) – kernel type; valid values: "julia", "python"

_python_kernel_name()

_python_kernel_display_name()

_do_make_kernel()

handle_kernelspec_install_process_finished(*retval*)

Handles case when the process for installing the kernel has finished.

Parameters

retval (*int*) – Process return value. 0: success, !0: failure

class spinetoolbox.widgets.kernel_editor.**MiniJuliaKernelEditor**(*parent*, *julia_exe*, *julia_project*)

Bases: *MiniKernelEditorBase*

A reduced version of KernelEditor that basically just takes care of installing one Julia kernel. The julia exe and project are passed in the constructor, then calling `make_kernel` starts the process.

Parameters

- **parent** (*QSettingsWidget*) – Toolbox settings widget
- **python_or_julia** (*str*) – kernel type; valid values: "julia", "python"

_julia_kernel_name()

_do_make_kernel()

handle_installkernel_process_finished(*retval*)

Checks whether or not the IJulia.installkernel process finished successfully.

Parameters

retval (*int*) – Process return value. 0: success, !0: failure

spinetoolbox.widgets.kernel_editor.**find_kernels**()

Returns a dictionary mapping kernel names to kernel paths.

spinetoolbox.widgets.kernel_editor.**find_python_kernels**()

Returns a dictionary of Python kernels. Keys are kernel_names, values are kernel paths.

spinetoolbox.widgets.kernel_editor.**find_julia_kernels**()

Returns a dictionary of Julia kernels. Keys are kernel_names, values are kernel paths.

spinetoolbox.widgets.kernel_editor.**find_unknown_kernels**()

Returns a dictionary of kernels that are neither Python nor Julia kernels.

spinetoolbox.widgets.kernel_editor.**format_event_message**(*msg_type*, *message*, *show_datetime=True*)

Formats message for the kernel editor text browser. This is a copy of helpers.format_event_message() but the colors have been edited for a text browser with a white background.

spinetoolbox.widgets.kernel_editor.**format_process_message**(*msg_type*, *message*)

Formats process message for the kernel editor text browser.

spinetoolbox.widgets.link_properties_widget

Link properties widget.

author

M. Marin (KTH)

date

27.11.2020

Module Contents

Classes

LinkPropertiesWidget	Widget for connection link properties.

class spinetoolbox.widgets.link_properties_widget.**LinkPropertiesWidget**(*toolbox*, *base_color=None*)

Bases: *spinetoolbox.widgets.properties_widget.PropertiesWidgetBase*

Widget for connection link properties.

Parameters

toolbox (ToolboxUI) – The toolbox instance where this widget should be embedded

set_link(*connection*)

Hooks the widget to given link, so that user actions are reflected in the link's filter configuration.

Parameters

connection (LoggingConnection) –

unset_link()

Releases the widget from any links.

_handle_write_index_value_changed(*value*)

_handle_use_datapackage_state_changed(*_state*)

_handle_use_memory_db_state_changed(*_state*)

_handle_purge_before_writing_state_changed(*_state*)

_open_purge_settings_dialog(*_=False*)

Opens the purge settings dialog.

_handle_purge_settings_changed()

Pushes a command that sets new purge settings onto undo stack.

_clean_up_purge_settings_dialog()

Cleans things related to purge settings dialog.

load_connection_options()

spinetoolbox.widgets.map_editor

An editor widget for editing a map type parameter values.

author

A. Soininen (VTT)

date

11.2.2020

Module Contents

Classes

MapEditor	A widget for editing maps.

class spinetoolbox.widgets.map_editor.**MapEditor**(*parent=None*)

Bases: PySide2.QtWidgets.QWidget

A widget for editing maps.

parent

Type
QWidget

_convert_leaves(_)

_show_table_context_menu(*position*)

Opens table context menu.

Parameters
position (*QPoint*) – menu's position

set_value(*value*)

Sets the parameter_value to be edited.

value()

Returns the parameter_value currently being edited.

open_value_editor(*index*)

Opens value editor dialog for given map model index.

Parameters
index (*QModelIndex*) – index

_open_header_editor(*column*)

spinetoolbox.widgets.map_value_editor

An editor dialog for map indexes and values.

author

A. Soininen (VTT)

date

2.11.2020

Module Contents

Classes

MapValueEditor	Dialog for editing parameter values in Map value editor.

class spinetoolbox.widgets.map_value_editor.**MapValueEditor**(*index*, *parent=None*)

Bases: *spinetoolbox.widgets.parameter_value_editor_base.ParameterValueEditorBase*

Dialog for editing parameter values in Map value editor.

Parameters

- **index** (*QModelIndex*) – an index to a parameter_value in parent_model
- **parent** (*QWidget, optional*) – a parent widget

_set_data(*value*)

See base class.

spinetoolbox.widgets.multi_tab_spec_editor

Contains the MultiTabSpecEditor class.

author

M. Marin (KTH)

date

12.12.2020

Module Contents

Classes

MultiTabSpecEditor	A main window that has a tab widget as its central widget.

class spinetoolbox.widgets.multi_tab_spec_editor.**MultiTabSpecEditor**(*toolbox*, *item_type*)

Bases: *spinetoolbox.widgets.multi_tab_window.MultiTabWindow*

A main window that has a tab widget as its central widget.

Parameters

- **qsettings** (*QSettings*) – Toolbox settings
- **settings_group** (*str*) – this window's settings group in qsettings

property new_tab_title

Title for new tabs.

_make_other()

Creates a new MultiTabWindow of this type.

Returns
new MultiTabWindow

Return type
MultiTabWindow

others()

List of other MultiTabWindows of the same type.

Returns
other MutliTabWindows windows

Return type
list of *MultiTabWindow*

_make_new_tab(**args*, ***kwargs*)

Creates a new tab.

Parameters

- ***args** – positional arguments neede to make a new tab
- ****kwargs** – keyword arguments needed to make a new tab

show_plus_button_context_menu(*global_pos*)

Opens a context menu for the tool bar.

Parameters
global_pos (*QPoint*) – menu position on screen

spinetoolbox.widgets.multi_tab_window

Contains the MultiTabWindow and TabBarPlus classes.

author

M. Marin (KTH)

date
12.12.2020

Module Contents

Classes

MultiTabWindow	A main window that has a tab widget as its central widget.
TabBarPlus	Tab bar that has a plus button floating to the right of the tabs.

class spinetoolbox.widgets.multi_tab_window.**MultiTabWindow**(*qsettings*, *settings_group*)

Bases: PySide2.QtWidgets.QMainWindow

A main window that has a tab widget as its central widget.

Parameters

- **qsettings** (*QSettings*) – Toolbox settings
- **settings_group** (*str*) – this window's settings group in qsettings

property accepting_new_tabs

property new_tab_title

Title for new tabs.

_tab_slots

abstract _make_other()

Creates a new MultiTabWindow of this type.

Returns
new MultiTabWindow

Return type
MultiTabWindow

abstract others()

List of other MultiTabWindows of the same type.

Returns
other MutliTabWindows windows

Return type
list of *MultiTabWindow*

abstract _make_new_tab(**args*, ***kwargs*)

Creates a new tab.

Parameters

- ***args** – positional arguments neede to make a new tab
- ****kwargs** – keyword arguments needed to make a new tab

abstract show_plus_button_context_menu(*global_pos*)

Opens a context menu for the tool bar.

Parameters
global_pos (*QPoint*) – menu position on screen

connect_signals()

Connects window's signals.

name()

Generates name based on the current tab and total tab count.

Returns
a name

Return type
str

all_tabs()

Iterates over tab contents widgets.

Yields
QWidget – tab contents widget

add_new_tab(**args*, ***kwargs*)

Creates a new tab and adds it at the end of the tab bar.

Parameters

- ***args** – parameters forwarded to `MutliTabWindow._make_new_tab()`
- ****kwargs** – parameters forwarded to `MultiTabwindow._make_new_tab()`

insert_new_tab(*index*, **args*, ***kwargs*)

Creates a new tab and inserts it at the given index.

Parameters

- **index** (*int*) – insertion point index
- ***args** – parameters forwarded to `MutliTabWindow._make_new_tab()`
- ****kwargs** – parameters forwarded to `MultiTabwindow._make_new_tab()`

_add_connect_tab(*tab*, *text*)

Appends a new tab and connects signals.

Parameters

- **tab** (*QWidget*) – tab contents widget
- **text** (*str*) – appended tab title

_insert_connect_tab(*index*, *tab*, *text*)

Inserts a new tab and connects signals.

Parameters

- **index** (*int*) – insertion point index
- **tab** (*QWidget*) – tab contents widget
- **text** (*str*) – inserted tab title

_remove_disconnect_tab(*index*)

Disconnects and removes a tab.

Parameters
index (*int*) – tab index

_connect_tab(*index*)

Connects signals from a tab contents widget.

Parameters
index (*int*) – tab index

_connect_tab_signals(*tab*)

Connects signals from a tab contents widget.

Parameters
tab (*QWidget*) – tab contents widget

Returns
True if signals were connected successfully, False otherwise

Return type
bool

_disconnect_tab_signals(*index*)

Disconnects signals from given tab.

Parameters
index (*int*) – tab index

Returns
True if signals were disconnected successfully, False otherwise

Return type
bool

_handle_tab_window_title_changed(*tab*, *title*)

Updates tab's title.

Parameters

- **tab** (*QWidget*) – tab's content widget
- **title** (*str*) – new tab title; if emtpy, one will be generated

_take_tab(*index*)

Removes a tab and returns its contents.

Parameters
index (*int*) – tab index

Returns
widget the tab was holding and tab's title

Return type
tuple

move_tab(*index*, *other=None*)

Moves a tab to another MultiTabWindow.

Parameters

- **index** (*int*) – tab index
- **other** (MultiTabWindow, *optional*) – target window; if None, creates a new window

detach(*index*, *hot_spot*, *offset=0*)

Detaches the tab at given index into another MultiTabWindow window and starts dragging it.

Parameters

- **index** (*int*) –
- **hot_spot** (*QPoint*) –
- **offset** (*int*) –

start_drag(*hot_spot*, *offset=0*)

Starts dragging a detached tab.

Parameters

- **hot_spot** (*QPoint*) – The anchor point of the drag in widget coordinates.
- **offset** (*int*) – Horizontal offset of the tab in the bar.

_frame_height()

Calculates the total 'thickness' of window frame in vertical direction.

Returns
frame height

Return type
int

timerEvent(*event*)

Performs the drag, i.e., moves the window with the mouse cursor. As soon as the mouse hovers the tab bar of another MultiTabWindow, reattaches it.

mouseReleaseEvent(*event*)

Stops the drag. This only happens when the detached tab is not reattached to another window.

reattach(*index*, *tab*, *text*)

Reattaches a tab that has been dragged over this window's tab bar.

Parameters

- **index** (*int*) – Index in this widget's tab bar where the detached tab has been dragged.
- **tab** (*QWidget*) – The widget in the tab being dragged.
- **text** (*str*) – The title of the tab.

_close_tab(*index*)

Closes the tab at index.

Parameters
index (*int*) – tab index

set_current_tab(*tab*)

Sets the tab that is shown on the window.

Parameters
tab (*QWidget*) – tab's contents widget

make_context_menu(*index*)

Creates a context menu for given tab.

Parameters
index (*int*) – tab index

Returns
context menu or None if tab was not found

Return type
QMenu

`restore_ui()`

Restore UI state from previous session.

`save_window_state()`

Save window state parameters (size, position, state) via QSettings.

`closeEvent(event)`

`class spinetoolbox.widgets.multi_tab_window.TabBarPlus(parent)`

Bases: `PySide2.QtWidgets.QTabBar`

Tab bar that has a plus button floating to the right of the tabs.

Parameters
parent (`MultiSpineDBEditor`) –

`plus_clicked`

`resizeEvent(event)`

Sets the dimension of the plus button. Also, makes the tab bar as wide as the parent.

`tabLayoutChange()`

`_move_plus_button()`

Places the plus button at the right of the last tab.

`mousePressEvent(event)`

Registers the position of the press, in case we need to detach the tab.

`mouseMoveEvent(event)`

Detaches a tab either if the user moves beyond the limits of the tab bar, or if it's the only one.

`_send_release_event(pos)`

Sends a mouse release event at given position in local coordinates. Called just before detaching a tab.

Parameters
pos (*QPoint*) –

`mouseReleaseEvent(event)`

`start_dragging(index)`

Stars dragging the given index. This happens when a detached tab is reattached to this bar.

Parameters
index (*int*) –

`index_under_mouse()`

Returns the index under the mouse cursor, or None if the cursor isn't over the tab bar. Used to check for drop targets.

Returns
int or NoneType

`contextMenuEvent(event)`

spinetoolbox.widgets.notification

Contains a notification widget.

author

P. Savolainen (VTT)

date

12.12.2019

Module Contents

Classes

Notification	Custom pop-up notification widget with fade-in and fade-out effect.
ButtonNotification	A notification with a button.
LinkNotification	A notification that may have a link.
ChangeNotifier	**param parent**

class spinetoolbox.widgets.notification.**Notification**(*parent*, *txt*, *anim_duration=500*, *life_span=None*, *word_wrap=True*, *corner=Qt.TopRightCorner*)

Bases: PySide2.QtWidgets.QFrame

Custom pop-up notification widget with fade-in and fade-out effect.

Parameters

- **parent** (*QWidget*) – Parent widget
- **txt** (*str*) – Text to display in notification
- **anim_duration** (*int*) – Duration of the animation in msecs
- **life_span** (*int*) – How long does the notification stays in place in msecs
- **word_wrap** (*bool*) –
- **corner** (*Qt.Corner*) –

opacity

show()

get_opacity()

opacity getter.

set_opacity(*op*)

opacity setter.

update_opacity(*value*)

Updates graphics effect opacity.

start_self_destruction()

Starts fade-out animation and closing of the notification.

enterEvent(*e*)

Pauses timer as the mouse hovers the notification.

leaveEvent(*e*)

Starts self destruction after the mouse leaves the notification.

remaining_time()

class spinetoolbox.widgets.notification.**ButtonNotification**(**args*, *button_text=''*, *button_slot=None*, ***kwargs*)

Bases: *Notification*

A notification with a button.

Parameters

- **parent** (*QWidget*) – Parent widget
- **txt** (*str*) – Text to display in notification
- **anim_duration** (*int*) – Duration of the animation in msecs
- **life_span** (*int*) – How long does the notification stays in place in msecs
- **word_wrap** (*bool*) –
- **corner** (*Qt.Corner*) –

class spinetoolbox.widgets.notification.**LinkNotification**(**args*, *open_link=None*, ***kwargs*)

Bases: *Notification*

A notification that may have a link.

Parameters

- **parent** (*QWidget*) – Parent widget
- **txt** (*str*) – Text to display in notification
- **anim_duration** (*int*) – Duration of the animation in msecs
- **life_span** (*int*) – How long does the notification stays in place in msecs
- **word_wrap** (*bool*) –
- **corner** (*Qt.Corner*) –

class spinetoolbox.widgets.notification.**ChangeNotifier**(*parent*, *undo_stack*, *settings*, *settings_key*, *corner=Qt.BottomRightCorner*)

Bases: PySide2.QtCore.QObject

Parameters

- **parent** (*QWidget*) –
- **undo_stack** (*QUndoStack*) –
- **settings** (*QSettings*) –
- **settings_key** (*str*) –

_push_notification(*index*)

spinetoolbox.widgets.open_project_widget

Contains a class for a widget that represents a 'Open Project Directory' dialog.

author

P. Savolainen (VTT)

date

1.11.2019

Module Contents

Classes

OpenProjectDialog	A dialog that lets user select a project to open either by choosing
CustomQFileSystemModel	Custom file system model.
DirValidator	

class spinetoolbox.widgets.open_project_widget.**OpenProjectDialog**(*toolbox*)

Bases: PySide2.QtWidgets.QDialog

A dialog that lets user select a project to open either by choosing an old .proj file or by choosing a project directory.

Parameters

toolbox (ToolboxUI) – QMainWindow instance

set_keyboard_shortcuts()

Creates keyboard shortcuts for the 'Root', 'Home', etc. buttons.

connect_signals()

Connects signals to slots.

expand_and_resize(*p*)

Expands, resizes, and scrolls the tree view to the current directory when the file model has finished loading the path. Slot for the file model's directoryLoaded signal. The directoryLoaded signal is emitted only if the directory has not been cached already. Note, that this is only used when the open project dialog is opened

Parameters

p (*str*) – Directory that has been loaded

validator_state_changed()

Changes the combobox border color according to the current state of the validator.

current_index_changed(*i*)

Combobox selection changed. This slot is processed when a new item is selected from the drop-down list. This is not processed when new item txt is QValidotor.Intermediate.

Parameters

i (*int*) – Selected row in combobox

current_changed(*current*, *previous*)

Processed when the current item in file system tree view has been changed with keyboard or mouse. Updates the text in combobox.

Parameters

- **current** (*QModelIndex*) – Currently selected index
- **previous** (*QModelIndex*) – Previously selected index

set_selected_path(*index*)

Sets the text in the combobox as the selected path in the file system tree view.

Parameters
index (*QModelIndex*) – The index which was mouse clicked.

combobox_text_edited(*text*)

Updates selected path when combobox text is edited. Note: pressing enter in combobox does not trigger this.

selection()

Returns the selected path from dialog.

go_root(*checked=False*)

Slot for the 'Root' button. Scrolls the treeview to show and select the user's root directory.

Note: We need to expand and scroll the tree view here after setCurrentIndex just in case the directory has been loaded already.

go_home(*checked=False*)

Slot for the 'Home' button. Scrolls the treeview to show and select the user's home directory.

go_documents(*checked=False*)

Slot for the 'Documents' button. Scrolls the treeview to show and select the user's documents directory.

go_desktop(*checked=False*)

Slot for the 'Desktop' button. Scrolls the treeview to show and select the user's desktop directory.

open_project(*index*)

Opens project if index contains a valid Spine Toolbox project. Slot for the mouse doubleClicked signal. Prevents showing the 'Not a valid spine toolbox project' notification if user just wants to collapse a directory.

Parameters
index (*QModelIndex*) – File model index which was double clicked

done(*r*)

Checks that selected path exists and is a valid Spine Toolbox directory when ok button is clicked or when enter is pressed without the combobox being in focus.

Parameters
r (*int*) –

static update_recents(*entry*, *qsettings*)

Adds a new entry to QSettings variable that remembers the five most recent project storages.

Parameters

- **entry** (*str*) – Abs. path to a directory that most likely contains other Spine Toolbox Projects as well. First entry is also used as the initial path for File->New Project dialog.
- **qsettings** (*QSettings*) – Toolbox qsettings object

static remove_directory_from_recents(*p*, *qsettings*)

Removes directory from the recent project storages.

Parameters

- **p** (*str*) – Full path to a project directory
- **qsettings** (*QSettings*) – Toolbox qsettings object

show_context_menu(*pos*)

Shows the context menu for the QCombobox with a 'Clear history' entry.

Parameters
pos (*QPoint*) – Mouse position

closeEvent(*event=None*)

Handles dialog closing.

Parameters
event (*QCloseEvent*) – Close event

class spinetoolbox.widgets.open_project_widget.**CustomQFileSystemModel**

Bases: PySide2.QtWidgets.QFileSystemModel

Custom file system model.

columnCount(*parent=QModelIndex()*)

Returns one.

class spinetoolbox.widgets.open_project_widget.**DirValidator**(*parent=None*)

Bases: PySide2.QtGui.QValidator

validate(*txt*, *pos*)

Returns Invalid if input is invalid according to this validator's rules, Intermediate if it is likely that a little more editing will make the input acceptable and Acceptable if the input is valid.

Parameters

- **txt** (*str*) – Text to validate
- **pos** (*int*) – Cursor position

Returns
Invalid, Intermediate, or Acceptable

Return type
QValidator.State

spinetoolbox.widgets.parameter_value_editor

An editor dialog for editing database (relationship) parameter values.

author

A. Soininen (VTT)

date
28.6.2019

Module Contents

Classes

ParameterValueEditor	Dialog for editing parameter values in Database editor.

class spinetoolbox.widgets.parameter_value_editor.**ParameterValueEditor**(*index*, *parent=None*, *plain=False*)

Bases: *spinetoolbox.widgets.parameter_value_editor_base.ParameterValueEditorBase*

Dialog for editing parameter values in Database editor.

Parameters

- **index** (*QModelIndex*) – an index to a parameter_value in parent_model
- **parent** (*QWidget, optional*) – a parent widget
- **plain** (*bool*) – if True, allow only plain value editing, otherwise allow all parameter types

_set_data(*value*)

See base class.

spinetoolbox.widgets.parameter_value_editor_base

A base for editor windows for editing parameter values.

author

A. Soininen (VTT)

date

2.11.2020

Module Contents

Classes

ValueType	Enum to identify value types that use different editors.
ParameterValueEditorBase	Dialog for editing parameter values.

Attributes

_SELECTORS

class spinetoolbox.widgets.parameter_value_editor_base.**ValueType**

Bases: enum.Enum

Enum to identify value types that use different editors.

PLAIN_VALUE

MAP

TIME_SERIES_FIXED_RESOLUTION

TIME_SERIES_VARIABLE_RESOLUTION

TIME_PATTERN

ARRAY

DATETIME

DURATION

spinetoolbox.widgets.parameter_value_editor_base.**_SELECTORS**

class spinetoolbox.widgets.parameter_value_editor_base.**ParameterValueEditorBase**(*index*, *editor_widgets*, *parent=None*)

Bases: PySide2.QtWidgets.QWidget

Dialog for editing parameter values.

The dialog takes an index and shows a specialized editor corresponding to the value type in a stack widget. The user can change the value type by changing the specialized editor using a combo box. When the dialog is closed the value from the currently shown specialized editor is written back to the given index.

Parameters

- **index** (*QModelIndex*) – an index to a parameter_value in parent_model
- **editor_widgets** (*dict*) – a mapping from *ValueType* to QWidget
- **parent** (*QWidget, optional*) – a parent widget

accept()

Saves the parameter_value shown in the currently selected editor widget to the database manager.

_change_parameter_type(*selector_index*)

Handles switching between value types.

Does a rude conversion between fixed and variable resolution time series. In other cases, a default 'empty' value is used.

Parameters
selector_index (*int*) – an index to the selector combo box

_select_editor(*value*)

Shows the editor widget corresponding to the given value type on the editor stack.

_use_default_editor(*message=None*)

Opens the default editor widget. Optionally, displays a warning dialog indicating the problem.

Parameters
message (*str, optional*) –

_use_editor(*value*, *value_type*)

Sets a value to edit on an editor widget.

Parameters

- **value** (*object*) – value to edit
- **value_type** (ValueType) – type of value

abstract _set_data(*value*)

Writes parameter value back to the model.

Parameters
value (*object*) – value to write

Returns
True if the operation was successful, False otherwise

Return type
bool

spinetoolbox.widgets.persistent_console_widget

Module Contents

Classes

_CustomLineEdit	
PersistentConsoleWidget	A widget to interact with a persistent process.
AnsiEscapeCodeHandler	

Functions

_ansi_color(code[, bright])

class spinetoolbox.widgets.persistent_console_widget.**_CustomLineEdit**(*console*)

Bases: PySide2.QtWidgets.QPlainTextEdit

property min_pos

property new_line_indent

reset(*current_prompt*)

new_line()

formatted_text()

raw_text()

set_raw_text(*text*)

_handle_text_changed()

Add indent to new lines.

_handle_cursor_position_changed()

Move cursor away from indent areas.

sizeHint()

keyPressEvent(*ev*)

class spinetoolbox.widgets.persistent_console_widget.**PersistentConsoleWidget**(*toolbox*, *key*, *language*, *owner=None*)

Bases: PySide2.QtWidgets.QPlainTextEdit

A widget to interact with a persistent process.

Parameters

- **toolbox** (ToolboxUI) –
- **key** (*tuple*) – persistent process identifier
- **language** (*str*) – for syntax highlighting and prompting, etc.
- **owner** (*ProjectItemBase, optional*) – console owner

property prompt

property owner_names

property _input_start_pos

_command_checked

_msg_available

_command_finished

_history_item_available

_completions_available

_restarted

_flush_needed

_FLUSH_INTERVAL = 200

_MAX_LINES_PER_SECOND = 2000

_MAX_LINES_PER_CYCLE

_MAX_LINES_COUNT = 2000

closeEvent(*ev*)

name()

Returns console name for display purposes.

focusInEvent(*ev*)

mouseMoveEvent(*ev*)

mousePressEvent(*ev*)

mouseReleaseEvent(*ev*)

scrollContentsBy(*dx*, *dy*)

_handle_contents_changed()

_handle_selection_changed()

_handle_cursor_position_changed()

_handle_update_request(*_rect*, *_dy*)

Move line edit to input start pos.

_move_line_edit()

_update_user_input()

_start_flush_timer()

_flush_text_buffer()

Inserts all text from buffer.

_make_prompt()

_make_prompt_block(*prompt=''*)

_insert_prompt(*prompt=''*)

_insert_stdin_text(*cursor*, *text*)

Inserts highlighted text.

Parameters

- **cursor** (*QTextCursor*) –
- **text** (*str*) –

_do_insert_stdin_text(*cursor*, *text*)

_insert_stdout_text(*cursor*, *text*)

Inserts ansi highlighted text.

Parameters

- **cursor** (*QTextCursor*) –
- **text** (*str*) –

_insert_text_before_prompt(*text*, *with_prompt=False*)

Inserts given text before the prompt. Used when adding input and output from external execution.

Parameters

text (*str*) –

_insert_text(*cursor*, *text*, *with_prompt*)

add_stdin(*data*)

Adds new prompt with data. Used when adding stdin from external execution.

Parameters
data (*str*) –

add_stdout(*data*)

Adds new line to stdout. Used when adding stdout from external execution.

Parameters
data (*str*) –

add_stderr(*data*)

Adds new line to stderr. Used when adding stderr from external execution.

Parameters
data (*str*) –

_get_current_text()

_get_prefix()

_highlight_current_input()

key_press_event(*ev*)

Handles key press event from line edit.

Returns
True if handled, False if not.

create_engine_manager()

Returns a new local or remote spine engine manager or an existing remote spine engine manager. Returns None if connecting to Spine Engine Server fails.

_issue_command(*text*)

Issues command.

Parameters
text (*str*) –

_do_check_command(*text*)

_handle_command_checked(*text*, *complete*)

Issues command.

Parameters
text (*str*) –

_do_issue_command(*text*)

_handle_msg_available(*msg_type*, *text*)

_handle_command_finished()

_move_history(*text*, *backwards*)

Moves history.

_do_move_history(*text*, *backwards*)

_display_history_item(*history_item*, *prefix*)

_autocomplete(*text*)

Autocompletes current text in the prompt (or output options if multiple matches).

Parameters
text (*str*) –

_do_autocomplete(*text*)

_display_completions(*text*, *prefix*, *completions*)

_restart_persistent(*_=False*)

Restarts underlying persistent process.

_do_restart_persistent()

_handle_restarted()

_interrupt_persistent(*_=False*)

Interrupts underlying persistent process.

_do_interrupt_persistent()

_extend_menu(*menu*)

Adds two more actions: Restart, and Interrupt.

contextMenuEvent(*ev*)

Reimplemented to extend menu with custom actions.

class spinetoolbox.widgets.persistent_console_widget.**AnsiEscapeCodeHandler**(*fg_color*, *bg_color*)

_make_default_format()

endFormatScope()

setFormatScope(*char_format*)

parse_text(*text*)

spinetoolbox.widgets.persistent_console_widget.**_ansi_color**(*code*, *bright=False*)

spinetoolbox.widgets.plain_parameter_value_editor

An editor widget for editing plain number database (relationship) parameter values.

author

A. Soininen (VTT)

date

28.6.2019

Module Contents

Classes

PlainParameterValueEditor	A widget to edit float or boolean type parameter values.

class spinetoolbox.widgets.plain_parameter_value_editor.**PlainParameterValueEditor**(*parent_widget=None*)

Bases: PySide2.QtWidgets.QWidget

A widget to edit float or boolean type parameter values.

parent_widget

a parent widget

Type
QWidget

_set_number_or_string_enabled(*on*)

set_value(*value*)

Sets the value to be edited in this widget.

value()

Returns the value currently being edited.

spinetoolbox.widgets.plot_canvas

A Qt widget to use as a matplotlib backend.

author

A. Soininen (VTT)

date
3.6.2019

Module Contents

Classes

PlotCanvas	A widget for plotting with matplotlib.

class spinetoolbox.widgets.plot_canvas.**PlotCanvas**(*parent=None*)

Bases: matplotlib.backends.backend_qt5agg.FigureCanvasQTAgg

A widget for plotting with matplotlib.

Parameters
parent (*QWidget*) – a parent widget

property axes

figure's axes

Type
`matplotlib.axes.Axes`

property legend_axes

figure's legend axes

Type
`matplotlib.axes.Axes`

spinetoolbox.widgets.plot_widget

A Qt widget showing a toolbar and a matplotlib plotting canvas.

author

A. Soininen (VTT)

date

27.6.2019

Module Contents

Classes

PlotWidget	A widget that contains a toolbar and a plotting canvas.
_PlotDataView	Custom QTableView class with copy and paste methods.
_PlotDataWidget	

Functions

prepare_plot_in_window_menu(menu)	Fills a given menu with available plot window names.

class spinetoolbox.widgets.plot_widget.**PlotWidget**(*parent=None*)

Bases: `PySide2.QtWidgets.QWidget`

A widget that contains a toolbar and a plotting canvas.

canvas

the plotting canvas

Type
PlotCanvas

original_xy_data

unmodified data on which the plots are based

Type
list of *XYData*

Parameters
parent (*QWidget*) – parent widget

plot_windows

A global list of plot windows.

closeEvent(*event*)

Removes the window from plot_windows and closes.

contextMenuEvent(*event*)

Shows plot context menu.

_get_plot_data()

Gathers plot data into a table.

Returns
data as table

Return type
list of list

copy_plot_data(*parent=None*, *document_name=''*)

Copies plot data to clipboard.

show_plot_data(*parent=None*, *document_name=''*)

Opens a separate window that shows the plot data.

add_legend()

Adds a legend to the plot's legend axes.

use_as_window(*parent_window*, *document_name*)

Prepares the widget to be used as a window and adds it to plot_windows list.

Parameters

- **parent_window** (*QWidget*) – a parent window
- **document_name** (*str*) – a string to add to the window title

static _unique_window_name(*document_name*)

Returns an unique identifier for a new plot window.

class spinetoolbox.widgets.plot_widget.**_PlotDataView**(*parent=None*)

Bases: *spinetoolbox.widgets.custom_qtableview.CopyPasteTableView*

Custom QTableView class with copy and paste methods.

contextMenuEvent(*event*)

class spinetoolbox.widgets.plot_widget.**_PlotDataWidget**(*rows*, *parent=None*)

Bases: PySide2.QtWidgets.QWidget

spinetoolbox.widgets.plot_widget.**prepare_plot_in_window_menu**(*menu*)

Fills a given menu with available plot window names.

Parameters
menu (*QMenu*) – menu to modify

spinetoolbox.widgets.plugin_manager_widgets

Contains PluginManager dialogs and widgets.

author

M. Marin (KTH)

date

21.2.2021

Module Contents

Classes

_InstallPluginModel	
_ManagePluginsModel	
InstallPluginDialog	Initialize class
ManagePluginsDialog	Initialize class

class spinetoolbox.widgets.plugin_manager_widgets.**_InstallPluginModel**

Bases: PySide2.QtGui.QStandardItemModel

data(*index*, *role=None*)

class spinetoolbox.widgets.plugin_manager_widgets.**_ManagePluginsModel**

Bases: *_InstallPluginModel*

flags(*index*)

class spinetoolbox.widgets.plugin_manager_widgets.**InstallPluginDialog**(*parent*)

Bases: PySide2.QtWidgets.QDialog

Initialize class

item_selected

populate_list(*names*)

_handle_search_text_changed(*_text*)

_filter_model()

_handle_ok_clicked(*_=False*)

_emit_item_selected(*index*)

_update_ok_button_enabled(*_selected*, *_deselected*)

class spinetoolbox.widgets.plugin_manager_widgets.**ManagePluginsDialog**(*parent*)

Bases: PySide2.QtWidgets.QDialog

Initialize class

item_removed

item_updated

populate_list(*names*)

_create_plugin_widget(*plugin_name*, *can_update*)

_emit_item_removed(*plugin_name*)

_emit_item_updated(*plugin_name*)

spinetoolbox.widgets.project_item_drag

Classes for custom QListView.

author

M. Marin (KTH)

date

14.11.2018

Module Contents

Classes

ProjectItemDragMixin	Custom class with dragging support.
NiceButton	
ProjectItemButtonBase	Custom class with dragging support.
ProjectItemButton	Custom class with dragging support.
ProjectItemSpecButton	Custom class with dragging support.
ShadeMixin	
ShadeProjectItemSpecButton	Custom class with dragging support.
ShadeButton	
_ChoppedIcon	
_ChoppedIconEngine	
ProjectItemSpecArray	An array of ProjectItemSpecButton that can be expanded/collapsed.

class spinetoolbox.widgets.project_item_drag.**ProjectItemDragMixin**(**args*, ***kwargs*)

Custom class with dragging support.

drag_about_to_start

mouseMoveEvent(*event*)

Start dragging action if needed

mouseReleaseEvent(*event*)

Forget drag start position

class spinetoolbox.widgets.project_item_drag.**NiceButton**(**args*, ***kwargs*)

Bases: PySide2.QtWidgets.QToolButton

setText(*text*)

set_orientation(*orientation*)

class spinetoolbox.widgets.project_item_drag.**ProjectItemButtonBase**(*toolbox*, *item_type*, *icon*, *parent=None*)

Bases: *ProjectItemDragMixin*, *NiceButton*

Custom class with dragging support.

set_colored_icons(*colored*)

_handle_drag_about_to_start()

mousePressEvent(*event*)

Register drag start position

abstract _make_mime_data_text()

class spinetoolbox.widgets.project_item_drag.**ProjectItemButton**(*toolbox*, *item_type*, *icon*, *parent=None*)

Bases: *ProjectItemButtonBase*

Custom class with dragging support.

double_clicked

_make_mime_data_text()

mouseDoubleClickEvent(*event*)

class spinetoolbox.widgets.project_item_drag.**ProjectItemSpecButton**(*toolbox*, *item_type*, *icon*, *spec_name=''*, *parent=None*)

Bases: *ProjectItemButtonBase*

Custom class with dragging support.

property spec_name

_make_mime_data_text()

contextMenuEvent(*event*)

mouseDoubleClickEvent(*event*)

class spinetoolbox.widgets.project_item_drag.**ShadeMixin**

paintEvent(*ev*)

class spinetoolbox.widgets.project_item_drag.**ShadeProjectItemSpecButton**(*toolbox*, *item_type*, *icon*, *spec_name=''*, *parent=None*)

Bases: *ShadeMixin*, *ProjectItemSpecButton*

Custom class with dragging support.

clone()

class spinetoolbox.widgets.project_item_drag.**ShadeButton**(**args*, ***kwargs*)

Bases: *ShadeMixin*, *NiceButton*

class spinetoolbox.widgets.project_item_drag.**_ChoppedIcon**(*icon*, *size*)

Bases: PySide2.QtGui.QIcon

update()

class spinetoolbox.widgets.project_item_drag.**_ChoppedIconEngine**(*icon*, *size*)

Bases: PySide2.QtGui.QIconEngine

update()

pixmap(*size*, *mode*, *state*)

class spinetoolbox.widgets.project_item_drag.**ProjectItemSpecArray**(*toolbox*, *model*, *item_type*, *icon*)

Bases: PySide2.QtWidgets.QToolBar

An array of ProjectItemSpecButton that can be expanded/collapsed.

Parameters

- **toolbox** (ToolboxUI) –
- **model** (FilteredSpecificationModel) –
- **item_type** (*str*) –
- **icon** (ColoredIcon) –

set_colored_icons(*colored*)

update()

_update_button_visible_icon_color()

set_color(*color*)

paintEvent(*ev*)

_get_first_chopped_index()

Returns the index of the first chopped action (chopped = not drawn because of space).

Returns
list(QAction) int or NoneType

_add_filling(*actions*, *ind*)

Adds a button to fill empty space after the last visible action.

Parameters

- **actions** (*list(QAction)*) – actions

- **ind** (*int or NoneType*) – index of the first chopped one or None if all are visible

_get_filling(*previous*)

Returns the position and size of the filling widget.

Parameters
previous (*QWidget*) – last visible widget

Returns
position x int: position y int: width int: height

Return type
int

_populate_extension_menu(*actions*, *ind*)

Populates extension menu with chopped actions.

Parameters

- **actions** (*list(QAction)*) – actions
- **ind** (*int or NoneType*) – index of the first chopped one or None if all are visible

showEvent(*ev*)

_update_button_geom(*orientation=None*)

Updates geometry of buttons given the orientation

Parameters
orientation (*Qt.Orientation*) –

_show_spec_form(*_checked=False*)

toggle_visibility(*_checked=False*)

set_visible(*visible*)

_insert_specs(*parent*, *first*, *last*)

_remove_specs(*parent*, *first*, *last*)

_remove_spec(*row*)

_reset_specs()

_add_spec(*row*)

spinetoolbox.widgets.properties_widget

Contains PropertiesWidgetBase.

author

M. Marin (ER)

date
20.01.2022

Module Contents

Classes

PropertiesWidgetBase	Properties widget base class.

class spinetoolbox.widgets.properties_widget.**PropertiesWidgetBase**(*toolbox*, *base_color=None*)

Bases: PySide2.QtWidgets.QWidget

Properties widget base class.

property fg_color

set_color_and_icon(*base_color*, *icon=None*)

eventFilter(*obj*, *ev*)

paintEvent(*ev*)

Paints background

spinetoolbox.widgets.rename_project_dialog

A widget for editing project name and description

Module Contents

Classes

RenameProjectDialog	Rename project dialog.

class spinetoolbox.widgets.rename_project_dialog.**RenameProjectDialog**(*toolbox*, *project*)

Bases: PySide2.QtWidgets.QDialog

Rename project dialog.

Parameters

- **toolbox** (ToolboxUI) – QMainWindow instance
- **project** (SpineToolboxProject) –

property name

property description

_set_ok_enabled()

accept()

spinetoolbox.widgets.report_plotting_failure

Functions to report failures in plotting to the user.

author

A. Soininen (VTT)

date

10.7.2019

Module Contents

Functions

report_plotting_failure(error, parent_widget)	Reports a PlottingError exception to the user.

spinetoolbox.widgets.report_plotting_failure.**report_plotting_failure**(*error*, *parent_widget*)

Reports a PlottingError exception to the user.

spinetoolbox.widgets.select_database_items

A widget and utilities to select database items.

Module Contents

Classes

SelectDatabaseItems	Widget that allows selecting database items.

Functions

add_check_boxes(check_boxes, checked_states, ...)	Adds check boxes to grid layout.
batch_set_check_state(boxes, checked)	Sets the checked state of multiple check boxes.

spinetoolbox.widgets.select_database_items.**add_check_boxes**(*check_boxes*, *checked_states*, *select_all_button*, *deselect_all_button*, *state_changed_slot*, *layout*)

Adds check boxes to grid layout.

Parameters

- **check_boxes** (*dict*) – mapping from label to QCheckBox
- **checked_states** (*dict*) – mapping from label to checked state boolean
- **select_all_button** (*QPushButton*) – the Select all button
- **deselect_all_button** (*QPushButton*) – the Deselect all button

- **state_changed_slot** (*Callable*) – slot to call when any checked state changes
- **layout** (*QGridLayout*) – target layout

spinetoolbox.widgets.select_database_items.**batch_set_check_state**(*boxes*, *checked*)

Sets the checked state of multiple check boxes.

Parameters

- **boxes** (*Iterable of QCheckBox*) – check boxes
- **checked** (*bool*) – checked state

class spinetoolbox.widgets.select_database_items.**SelectDatabaseItems**(*checked_states=None*, *parent=None*)

Bases: PySide2.QtWidgets.QWidget

Widget that allows selecting database items.

Parameters

- **checked_states** (*dict, optional*) – mapping from item name to check state boolean
- **parent** (*QWidget*) – parent widget

checked_state_changed

COLUMN_COUNT = 3

_DATA_ITEMS = ['object', 'relationship', 'entity_group', 'parameter_value', 'entity_metadata',...

checked_states()

Collects the checked states of database items.

Returns
mapping from item name to checked state boolean

Return type
dict

any_checked()

Checks if any of the check boxes is checked.

Returns
True if any check box is checked, False otherwise

Return type
bool

_select_data_items(_=*False*)

Checks all data items.

spinetoolbox.widgets.settings_widget

Widget for controlling user settings.

author

P. Savolainen (VTT)

date

17.1.2018

Module Contents

Classes

SettingsWidgetBase	**param qsettings** Toolbox settings
SpineDBEditorSettingsMixin	
SpineDBEditorSettingsWidget	A widget to change user's preferred settings, but only for the Spine db editor.
SettingsWidget	A widget to change user's preferred settings.

Functions

_get_python_kernel_name_by_exe(python_exe)	Returns a kernel name corresponding to given python exe, or an empty string if none available.
_get_julia_kernel_name_by_env(julia_exe, julia_project)	Returns a kernel name corresponding to given julia exe and project, or an empty string if none available.
_samefile(a, b)	

class spinetoolbox.widgets.settings_widget.**SettingsWidgetBase**(*qsettings*)

Bases: PySide2.QtWidgets.QWidget

Parameters

qsettings (*QSettings*) – Toolbox settings

property qsettings

connect_signals()

Connect signals.

keyPressEvent(*e*)

Close settings form when escape key is pressed.

Parameters

e (*QKeyEvent*) – Received key press event.

mousePressEvent(*e*)

Save mouse position at the start of dragging.

Parameters
e (*QMouseEvent*) – Mouse event

mouseReleaseEvent(*e*)

Save mouse position at the end of dragging.

Parameters
e (*QMouseEvent*) – Mouse event

mouseMoveEvent(*e*)

Moves the window when mouse button is pressed and mouse cursor is moved.

Parameters
e (*QMouseEvent*) – Mouse event

update_ui()

Updates UI to reflect current settings. Called when the user choses to cancel their changes. Undoes all temporary UI changes that resulted from the user playing with certain settings.

save_settings()

Gets selections and saves them to persistent memory.

update_ui_and_close(*checked=False*)

Updates UI to reflect current settings and close.

save_and_close(*checked=False*)

Saves settings and close.

class spinetoolbox.widgets.settings_widget.**SpineDBEditorSettingsMixin**

connect_signals()

Connect signals.

read_settings()

Read saved settings from app QSettings instance and update UI to display them.

save_settings()

Get selections and save them to persistent memory.

update_ui()

set_auto_expand_objects(*checked=False*)

set_merge_dbs(*checked=False*)

class spinetoolbox.widgets.settings_widget.**SpineDBEditorSettingsWidget**(*multi_db_editor*)

Bases: *SpineDBEditorSettingsMixin*, *SettingsWidgetBase*

A widget to change user's preferred settings, but only for the Spine db editor.

Initialize class.

property db_mngr

show()

class spinetoolbox.widgets.settings_widget.**SettingsWidget**(*toolbox*)

Bases: *SpineDBEditorSettingsMixin*, *SettingsWidgetBase*

A widget to change user's preferred settings.

Parameters
toolbox (ToolboxUI) – Parent widget.

property db_mngr

connect_signals()

Connect signals.

_update_python_widgets_enabled(*state*)

_update_julia_widgets_enabled(*state*)

_update_remote_execution_page_widget_status(*state*)

Enables or disables widgets on Remote Execution page, based on the state of remote execution enabled check box.

_show_install_julia_wizard()

_show_add_up_spine_opt_wizard()

browse_gams_button_clicked(*checked=False*)

Calls static method that shows a file browser for selecting a Gams executable.

browse_julia_button_clicked(*checked=False*)

Calls static method that shows a file browser for selecting a Julia path.

browse_julia_project_button_clicked(*checked=False*)

Calls static method that shows a folder browser for selecting a Julia project.

browse_python_button_clicked(*checked=False*)

Calls static method that shows a file browser for selecting a Python interpreter.

browse_conda_button_clicked(*checked=False*)

Calls static method that shows a file browser for selecting a Conda executable.

browse_certificate_directory_clicked(_)

Calls static method that shows a file browser for selecting the security folder for Engine Server.

show_python_kernel_editor(*checked=False*)

Opens kernel editor, where user can make a kernel for the Python Console.

python_kernel_editor_closed(*ret_code*)

Catches the selected Python kernel name when the editor is closed.

show_julia_kernel_editor(*checked=False*)

Opens kernel editor, where user can make a kernel the Julia Console.

julia_kernel_editor_closed(*ret_code*)

Catches the selected Julia kernel name when the editor is closed.

browse_work_path(*checked=False*)

Open file browser where user can select the path to wanted work directory.

show_color_dialog(*checked=False*)

Let user pick the bg color.

Parameters
checked (*boolean*) – Value emitted with clicked signal

update_bg_color()

Set tool button icon as the selected color and update Design View scene background color.

update_scene_bg(*checked=False*)

Draw background on scene depending on radiobutton states.

Parameters
checked (*boolean*) – Toggle state

update_links_geometry(*checked=False*)

update_items_path(*checked=False*)

set_toolbar_colored_icons(*checked=False*)

_update_properties_widget(*_checked=False*)

read_settings()

Read saved settings from app QSettings instance and update UI to display them.

_read_engine_settings()

Reads Engine settings and sets the corresponding UI elements.

save_settings()

Get selections and save them to persistent memory. Note: On Linux, True and False are saved as boolean values into QSettings. On Windows, booleans and integers are saved as strings. To make it consistent, we should use strings.

_save_engine_settings()

Stores Engine settings to application settings.

Returns
True if settings were stored successfully, False otherwise

Return type
bool

_get_julia_settings()

set_work_directory(*new_work_dir*)

Sets new work directory.

Parameters
new_work_dir (*str*) – Possibly a new work directory

update_ui()

Updates UI to reflect current settings. Called when the user choses to cancel their changes. Undoes all temporary UI changes that resulted from the user playing with certain settings.

_edit_remote_host(*new_text*)

Prepends host line edit with the protocol for user convenience.

Parameters
new_text (*str*) – Text in the line edit after user has entered a character

closeEvent(*ev*)

spinetoolbox.widgets.settings_widget.**_get_python_kernel_name_by_exe**(*python_exe*)

Returns a kernel name corresponding to given python exe, or an empty string if none available.

Parameters
python_exe (*str*) –

Returns
str

spinetoolbox.widgets.settings_widget.**_get_julia_kernel_name_by_env**(*julia_exe*, *julia_project*)

Returns a kernel name corresponding to given julia exe and project, or an empty string if none available.

Parameters

- **julia_exe** (*str*) –
- **julia_project** (*str*) –

Returns
str

spinetoolbox.widgets.settings_widget.**_samefile**(*a*, *b*)

spinetoolbox.widgets.statusbars

Functions to make and handle QStatusBars.

Module Contents

Classes

MainStatusBar	A status bar for the main toolbox window.

class spinetoolbox.widgets.statusbars.**MainStatusBar**(*toolbox*)

Bases: PySide2.QtWidgets.QStatusBar

A status bar for the main toolbox window.

Parameters
toolbox (ToolboxUI) –

_ALL_RUNS = All executions

_populate_executions_menu()

reset_executions_button_text()

_select_execution(*action*)

spinetoolbox.widgets.time_pattern_editor

An editor widget for editing a time pattern type (relationship) parameter values.

author

A. Soininen (VTT)

date

28.6.2019

Module Contents

Classes

TimePatternEditor	A widget for editing time patterns.

class spinetoolbox.widgets.time_pattern_editor.**TimePatternEditor**(*parent=None*)

Bases: PySide2.QtWidgets.QWidget

A widget for editing time patterns.

Parameters

parent (*QWidget*) – parent widget

_show_table_context_menu(*position*)

Opens the table's context menu.

Parameters

position (*QPoint*) – menu's position on the table

set_value(*value*)

Sets the parameter_value to be edited.

value()

Returns the parameter_value currently being edited.

_open_header_editor(*column*)

spinetoolbox.widgets.time_series_fixed_resolution_editor

Contains logic for the fixed step time series editor widget.

author

A. Soininen (VTT)

date

14.6.2019

Module Contents

Classes

TimeSeriesFixedResolutionEditor	A widget for editing time series data with a fixed time step.

Functions

_resolution_to_text(resolution)	Converts a list of durations into a string of comma-separated durations.
_text_to_resolution(text)	Converts a comma-separated string of durations into a resolution array.

spinetoolbox.widgets.time_series_fixed_resolution_editor.**_resolution_to_text**(*resolution*)

Converts a list of durations into a string of comma-separated durations.

spinetoolbox.widgets.time_series_fixed_resolution_editor.**_text_to_resolution**(*text*)

Converts a comma-separated string of durations into a resolution array.

class spinetoolbox.widgets.time_series_fixed_resolution_editor.**TimeSeriesFixedResolutionEditor**(*parent=No*

Bases: PySide2.QtWidgets.QWidget

A widget for editing time series data with a fixed time step.

Parameters
parent (*QWidget*) – a parent widget

_resolution_changed()

Updates the models after resolution change.

_show_table_context_menu(*position*)

Shows the table's context menu.

Parameters
position (*QPoint*) – menu's position in table view's coordinates

_select_date(*selected_date*)

set_value(*value*)

Sets the parameter_value for editing in this widget.

_show_calendar()

_start_time_changed()

Updates the model due to start time change.

_update_plot(*topLeft=None*, *bottomRight=None*, *roles=None*)

Updated the plot.

value()

Returns the parameter_value currently being edited.

_open_header_editor(*column*)

spinetoolbox.widgets.time_series_variable_resolution_editor

Contains logic for the variable resolution time series editor widget.

author

A. Soininen (VTT)

date

31.5.2019

Module Contents

Classes

TimeSeriesVariableResolutionEditor	A widget for editing variable resolution time series data.

class spinetoolbox.widgets.time_series_variable_resolution_editor.**TimeSeriesVariableResolutionEditor**(*par*

Bases: `PySide2.QtWidgets.QWidget`

A widget for editing variable resolution time series data.

Parameters

parent (*QWidget*) – a parent widget

_show_table_context_menu(*position*)

Shows the table's context menu.

Parameters

position (*QPoint*) – menu's position on the table

set_value(*value*)

Sets the time series being edited.

_update_plot(*topLeft=None*, *bottomRight=None*, *roles=None*)

Updates the plot widget.

value()

Return the time series currently being edited.

_open_header_editor(*column*)

spinetoolbox.widgets.toolbars

Functions to make and handle QToolBars.

author

P. Savolainen (VTT)

date

19.1.2018

Module Contents

Classes

ToolBar	Base class for Toolbox toolbars.
PluginToolBar	A plugin toolbar.
MainToolBar	The main application toolbar: Items | Execute
PaddingLabel	

class spinetoolbox.widgets.toolbars.**ToolBar**(*name*, *toolbox*)

Bases: PySide2.QtWidgets.QToolBar

Base class for Toolbox toolbars.

Parameters

- **name** (*str*) – toolbar's name
- **toolbox** (ToolboxUI) – Toolbox main window

abstract set_color(*color*)

Sets toolbar's background color.

Parameters
color (*QColor*) – background color

set_project_actions_enabled(*enabled*)

Enables or disables project related actions.

Parameters
enabled (*bool*) – True to enable actions, False to disable

class spinetoolbox.widgets.toolbars.**PluginToolBar**(*name*, *parent*)

Bases: *ToolBar*

A plugin toolbar.

Parameters
parent (ToolboxUI) – QMainWindow instance

setup(*plugin_specs*, *disabled_names*)

Sets up the toolbar.

Parameters

- **plugin_specs** (*dict*) – mapping from specification name to specification
- **disabled_names** (*Iterable of str*) – specifications that should be disabled

set_color(*color*)

Sets toolbar's background color.

Parameters
color (*QColor*) – background color

_update_spec_button_name(*old_name*, *new_name*)

class spinetoolbox.widgets.toolbars.**MainToolBar**(*execute_project_action*, *execute_selection_action*, *stop_execution_action*, *parent*)

Bases: *ToolBar*

The main application toolbar: Items | Execute

Parameters

- **execute_project_action** (*QAction*) – action to execute project
- **execute_selection_action** (*QAction*) – action to execute selected items
- **stop_execution_action** (*QAction*) – action to stop execution
- **parent** (ToolboxUI) – QMainWindow instance

_SEPARATOR = ;;

set_project_actions_enabled(*enabled*)

Enables or disables project related actions.

Parameters
enabled (*bool*) – True to enable actions, False to disable

set_color(*color*)

Sets toolbar's background color.

Parameters
color (*QColor*) – background color

setup()

add_project_item_buttons()

_add_project_item_button(*item_type*, *factory*, *colored*)

set_colored_icons(*colored*)

_make_tool_button(*icon*, *text*, *slot*, *tip=None*)

Makes a new tool button and adds it to the toolbar.

Parameters

- **icon** (*QIcon*) – button's icon
- **text** (*str*) – button's text
- **slot** (*Callable*) – slot where to connect button's clicked signal
- **tip** (*str*) – button's tooltip

Returns
created button

Return type
QToolButton

_add_tool_button(*button*)

Adds a button to the toolbar.

Parameters
button (*QToolButton*) – button to add

add_execute_buttons()

Adds project execution buttons to the toolbar.

dragLeaveEvent(*event*)

dragEnterEvent(*event*)

dragMoveEvent(*event*)

dropEvent(*event*)

_update_drop_actions(*event*)

Updates source and target actions for drop operation:

Parameters
event (*QDragMoveEvent*) –

paintEvent(*ev*)

Draw a line as drop indicator.

_drop_line()

icon_ordering()

class spinetoolbox.widgets.toolbars.**PaddingLabel**(**args*, ***kwargs*)

Bases: PySide2.QtWidgets.QLabel

20.1.2 Submodules

spinetoolbox.__main__

Spine Toolbox application main file.

author

P. Savolainen (VTT)

date

14.12.2017

Module Contents

spinetoolbox.__main__.**return_code**

spinetoolbox.config

Application constants and style sheets

author

P. Savolainen (VTT)

date

2.1.2018

Module Contents

spinetoolbox.config.**LATEST_PROJECT_VERSION = 8**

spinetoolbox.config.**REQUIRED_SPINE_OPT_VERSION = 0.5.3**

spinetoolbox.config.**INVALID_CHARS = ['<', '>', ':', '"', '/', '\\', '|', '?', '*', '.']**

spinetoolbox.config.**INVALID_FILENAME_CHARS = ['<', '>', ':', '"', '/', '\\', '|', '?', '*']**

spinetoolbox.config.**_frozen**

spinetoolbox.config.**_path_to_executable**

spinetoolbox.config.**APPLICATION_PATH**

spinetoolbox.config.**_program_root**

spinetoolbox.config.**DEFAULT_WORK_DIR**

spinetoolbox.config.**DOCUMENTATION_PATH**

spinetoolbox.config.**ONLINE_DOCUMENTATION_URL = https://spine-toolbox.readthedocs.io/en/master/**

spinetoolbox.config.**PLUGINS_PATH**

spinetoolbox.config.**PLUGIN_REGISTRY_URL = https://spine-project.github.io/PluginRegistry/registry.json**

spinetoolbox.config.**JUPYTER_KERNEL_TIME_TO_DEAD = 8.0**

spinetoolbox.config.**PROJECT_FILENAME = project.json**

spinetoolbox.config.**PROJECT_LOCAL_DATA_DIR_NAME = local**

spinetoolbox.config.**PROJECT_LOCAL_DATA_FILENAME = project_local_data.json**

spinetoolbox.config.**SPECIFICATION_LOCAL_DATA_FILENAME = specification_local_data.json**

spinetoolbox.config.**PROJECT_ZIP_FILENAME = project_package**

spinetoolbox.config.**STATUSBAR_SS = QStatusBar{background-color: #EBEBE0; border-width: 1px; border-color: gray; border-style: groove;}**

spinetoolbox.config.**BG_COLOR = #19232D**

spinetoolbox.config.**FG_COLOR = #F0F0F0**

spinetoolbox.config.**SETTINGS_SS**

spinetoolbox.config.**ICON_BACKGROUND = qlineargradient(x1: 1, y1: 1, x2: 0, y2: 0, stop: 0 #cce0ff, stop: 1 #66a1ff);**

spinetoolbox.config.**ICON_TOOLBAR_SS**

spinetoolbox.config.**TEXTBROWSER_SS**

spinetoolbox.config.**MAINWINDOW_SS**

spinetoolbox.config.**PIVOT_TABLE_HEADER_COLOR = #efefef**

spinetoolbox.execution_managers

Classes to manage tool instance execution in various forms.

author

P. Savolainen (VTT)

date

1.2.2018

Module Contents

Classes

ExecutionManager	Base class for all tool instance execution managers.
QProcessExecutionManager	Class to manage tool instance execution using a PySide2 QProcess.

class spinetoolbox.execution_managers.**ExecutionManager**(*logger*)

Bases: PySide2.QtCore.QObject

Base class for all tool instance execution managers.

Class constructor.

Parameters

logger (LoggerInterface) – a logger instance

execution_finished

abstract start_execution(*workdir=None*)

Starts the execution.

Parameters

workdir (*str*) – Work directory

abstract stop_execution()

Stops the execution.

class spinetoolbox.execution_managers.**QProcessExecutionManager**(*logger*, *program=''*, *args=None*, *silent=False*, *semisilent=False*)

Bases: *ExecutionManager*

Class to manage tool instance execution using a PySide2 QProcess.

Class constructor.

Parameters

- **logger** (LoggerInterface) – a logger instance
- **program** (*str*) – Path to program to run in the subprocess (e.g. julia.exe)
- **args** (*list, optional*) – List of argument for the program (e.g. path to script file)
- **silent** (*bool*) – Whether or not to emit logger msg signals
- **semisilent** (*bool*) – If True, show Process Log messages

program()

Program getter method.

args()

Program argument getter method.

start_execution(*workdir=None*)

Starts the execution of a command in a QProcess.

Parameters
workdir (*str, optional*) – Work directory

wait_for_process_finished(*msecs=30000*)

Wait for subprocess to finish.

Parameters
msecs (*int*) – Timeout in milliseconds

Returns
True if process finished successfully, False otherwise

process_started()

Run when subprocess has started.

on_state_changed(*new_state*)

Runs when QProcess state changes.

Parameters
new_state (*int*) – Process state number (`QProcess::ProcessState`)

on_process_error(*process_error*)

Runs if there is an error in the running QProcess.

Parameters
process_error (*int*) – Process error number (`QProcess::ProcessError`)

teardown_process()

Tears down the QProcess in case a QProcess.ProcessError occurred. Emits execution_finished signal.

stop_execution()

See base class.

on_process_finished(*exit_code*, *exit_status*)

Runs when subprocess has finished.

Parameters

- **exit_code** (*int*) – Return code from external program (only valid for normal exits)
- **exit_status** (*int*) – Crash or normal exit (`QProcess::ExitStatus`)

on_ready_stdout()

Emit data from stdout.

on_ready_stderr()

Emit data from stderr.

spinetoolbox.headless

Contains facilities to open and execute projects without GUI.

authors

A. Soininen (VTT)

date

29.4.2020

Module Contents

Classes

HeadlessLogger	A `LoggerInterface` compliant logger that uses Python's standard logging facilities.
ModifiableProject	A simple project that is available for modification script.
ActionsWithProject	A 'task' which opens Toolbox project and operates on it.
Status	Status codes returned from headless execution.

Functions

headless_main(args)	Executes a project using `QCoreApplication`.
open_project(project_dict, project_dir, logger)	Opens a project.
_specification_dicts(project_dict, project_dir, logger)	Loads project item specification dictionaries.

class spinetoolbox.headless.**HeadlessLogger**

Bases: `PySide2.QtCore.QObject`

A `LoggerInterface` compliant logger that uses Python's standard logging facilities.

msg

Emits a notification message.

msg_success

Emits a message on success

msg_warning

Emits a warning message.

msg_error

Emits an error message.

msg_proc

Emits a message originating from a subprocess (usually something printed to stdout).

msg_proc_error

Emits an error message originating from a subprocess (usually something printed to stderr).

information_box

Requests an 'information message box' (e.g. a message window) to be opened with a given title and message.

error_box

Requests an 'error message box' to be opened with a given title and message.

_log_message(*message*)

Prints an information message.

_log_warning(*message*)

Prints a warning message.

_log_error(*message*)

Prints an error message.

_show_information_box(*title*, *message*)

Prints an information message with a title.

_show_error_box(*title*, *message*)

Prints an error message with a title.

_print(*message*, *out_stream*)

Filters HTML tags from message before printing it to given file.

class spinetoolbox.headless.**ModifiableProject**(*project_dir*, *items_dict*, *connection_dicts*)

A simple project that is available for modification script.

Parameters

- **project_dir** (*Path*) – project directory
- **items_dict** (*dict*) – project item dictionaries
- **connection_dicts** (*list of dict*) – connection dictionaries

property project_dir

find_connection(*source_name*, *destination_name*)

Searches for a connection between given items.

Parameters

- **source_name** (*str*) – source item's name
- **destination_name** (*str*) – destination item's name

Returns

connection instance or None if there is no connection

Return type

Connection

find_item(*name*)

Searches for a project item.

Parameters

name (*str*) – item's name

Returns

item dict or None if no such item exists

Return type
dict

`items_to_dict()`

Stores project items back to dictionaries.

Returns
item dictionaries

Return type
dict

`connections_to_dict()`

Stores connections back to dictionaries.

Returns
connection dictionaries

Return type
list of dict

`class spinetoolbox.headless.ActionsWithProject(args, startup_event_type, parent)`

Bases: `PySide2.QtCore.QObject`

A 'task' which opens Toolbox project and operates on it.

The execution of this task is triggered by sending it a 'startup' QEvent using e.g. QCoreApplication.postEvent()

Parameters

- **args** (*argparse.Namespace*) – parsed command line arguments
- **startup_event_type** (*int*) – expected type id for the event that starts this task
- **parent** (*QObject*) – a parent object

`_start`

A private signal to actually start execution. Not to be used directly. Post a startup event instead.

`_dags()`

`_execute()`

Executes this task.

`_open_project()`

Opens a project.

Returns
status code

Return type
Status

`_check_project_version(project_dict)`

Checks project dict version.

Parameters
project_dict (*dict*) – project dict

Returns
status code

Return type
Status

_exec_mod_script()

Executes project modification script given in command line arguments.

Returns
status code

Return type
Status

_execute_project()

Executes all DAGs in a project.

Returns
status code

Return type
Status

_process_engine_event(*event_type*, *data*)

event(*e*)

_handle_node_execution_started(*data*)

Starts collecting messages from given node.

Parameters
data (*dict*) – execution start data

_handle_node_execution_finished(*data*)

Prints messages for finished nodes.

Parameters
data (*dict*) – execution end data

_handle_event_msg(*data*)

Stores event messages for later printing.

Parameters
data (*dict*) – event message data

_handle_process_msg(*data*)

Stores process messages for later printing.

Parameters
data (*dict*) – process message data

_handle_standard_execution_msg(*data*)

Handles standard execution messages.

Currently, these messages are ignored.

Parameters
data (*dict*) – execution message data

_handle_persistent_execution_msg(*data*)

Handles persistent execution messages.

Parameters
data (*dict*) – execution message data

_handle_kernel_execution_msg(*data*)

Handles kernel messages.

Currently, these messages are ignored.

Parameters
data (*dict*) – message data

spinetoolbox.headless.**headless_main**(*args*)

Executes a project using QCoreApplication.

Parameters
args (*argparser.Namespace*) – parsed command line arguments.

Returns
exit status code; 0 for success, everything else for failure

Return type
int

spinetoolbox.headless.**open_project**(*project_dict*, *project_dir*, *logger*)

Opens a project.

Parameters

- **project_dict** (*dict*) – a serialized project dictionary
- **project_dir** (*Path*) – path to a directory containing the .spinetoolbox dir
- **logger** (LoggerInterface) – a logger

Returns
item dicts, specification dicts, connection dicts, jump dicts and a DagHandler object

Return type
tuple

spinetoolbox.headless.**_specification_dicts**(*project_dict*, *project_dir*, *logger*)

Loads project item specification dictionaries.

Parameters

- **project_dict** (*dict*) – a serialized project dictionary
- **project_dir** (*str*) – path to a directory containing the .spinetoolbox dir
- **logger** (LoggerInterface) – a logger

Returns
a mapping from item type to a list of specification dicts

Return type
dict

class spinetoolbox.headless.**Status**

Bases: enum.IntEnum

Status codes returned from headless execution.

Initialize self. See help(type(self)) for accurate signature.

OK = 0

ERROR = 1

ARGUMENT_ERROR = 2

spinetoolbox.helpers

General helper functions and classes.

authors

P. Savolainen (VTT)

date

10.1.2018

Module Contents

Classes

LinkType	Graphics scene's link types.
IconListManager	A class to manage icons for icon list widgets.
TransparentIconEngine	Specialization of QIconEngine with transparent background.
CharIconEngine	Specialization of QIconEngine used to draw font-based icons.
ColoredIcon	
ColoredIconEngine	
ProjectDirectoryIconProvider	QFileIconProvider that provides a Spine icon to the
ChildCyclingKeyPressFilter	Event filter class for catching next and previous child key presses.
QuietLogger	
SignalWaiter	A 'traffic light' that allows waiting for a signal to be emitted in another thread.
CustomSyntaxHighlighter	
FetchParent	
ItemTypeFetchParent	
HTMLTagFilter	HTML tag filter.

Functions

home_dir()	Returns user's home dir
format_log_message(msg_type, message[, show_datetime])	Adds color tags and optional time stamp to message.
busy_effect(func)	Decorator to change the mouse cursor to 'busy' while a function is processed.
create_dir(base_path[, folder, verbosity])	Create (input/output) directories recursively.
rename_dir(old_dir, new_dir, toolbox, box_title)	Renames directory. Called by `ProjectItemModel.set_item_name()`
open_url(url)	Opens the given url in the appropriate Web browser for the user's desktop environment,
set_taskbar_icon()	Set application icon to Windows taskbar.
supported_img_formats()	Checks if reading .ico files is supported.
pyside2_version_check()	Check that PySide2 version is 5.14 or 5.15.
get_datetime(show[, date])	Returns date and time string for appending into Event Log messages.
copy_files(src_dir, dst_dir[, includes, excludes])	Function for copying files. Does not copy folders.
erase_dir(path[, verbosity])	Deletes a directory and all its contents without prompt.
recursive_overwrite(logger, src, dst[, ignore, silent])	Copies everything from source directory to destination directory recursively.
tuple_itemgetter(itemgetter_func, num_indexes)	Change output of itemgetter to always be a tuple even for a single index.
format_string_list(str_list)	Returns a html unordered list from the given list of strings.
rows_to_row_count_tuples(rows)	Breaks a list of rows into a list of (row, count) tuples to corresponding
object_icon(display_icon)	Creates and returns a QIcon corresponding to display_icon.
color_pixmap(pixmap, color)	
make_icon_id(icon_code, color_code)	Takes icon and color codes, and return equivalent integer.
interpret_icon_id(display_icon)	Takes a display icon id and returns an equivalent tuple of icon and color code.
default_icon_id()	Creates a default icon id.
ensure_window_is_on_screen(window, size)	Checks if window is on screen and if not, moves and resizes it to make it visible on the primary screen.
first_non_null(s)	Returns the first element in Iterable s that is not None.
get_save_file_name_in_last_dir(qsettings, key, parent, ...)	Calls QFileDialog.getSaveFileName in the directory that was selected last time the dialog was accepted.
get_open_file_name_in_last_dir(qsettings, key, parent, ...)	
try_number_from_string(text)	Tries to convert a string to integer or float.
focused_widget_has_callable(parent, callable_name)	Returns True if the currently focused widget or one of its ancestors has the given callable.
call_on_focused_widget(parent, callable_name)	Calls the given callable on the currently focused widget or one of its ancestors.
select_gams_executable(parent, line_edit)	Opens file browser where user can select a Gams executable (i.e. gams.exe on Windows).

continues on next page

Table 1 – continued from previous page

select_julia_executable(parent, line_edit)	Opens file browser where user can select a Julia executable (i.e. julia.exe on Windows).
select_julia_project(parent, line_edit)	Shows file browser and inserts selected julia project dir to give line_edit.
select_python_interpreter(parent, line_edit)	Opens file browser where user can select a python interpreter (i.e. python.exe on Windows).
select_conda_executable(parent, line_edit)	Opens file browser where user can select a conda executable.
select_certificate_directory(parent, line_edit)	Shows file browser and inserts selected certificate directory to given line edit.
file_is_valid(parent, file_path, msgbox_title[, ...])	Checks that given path is not a directory and it's a file that actually exists.
dir_is_valid(parent, dir_path, msgbox_title)	Checks that given path is a directory. Needed in
make_settings_dict_for_engine(app_settings)	Converts Toolbox settings to a dictionary acceptable by Engine.
make_icon_background(color)	
make_icon_toolbar_ss(color)	
color_from_index(i, count[, base_hue, saturation])	
unique_name(prefix, existing)	Creates a unique name in the form "prefix X" where X is a number.
get_upgrade_db_promt_text(url, current, expected)	
parse_specification_file(spec_path, logger)	Parses specification file.
load_specification_from_file(spec_path, ...)	Returns an Item specification from a definition file.
specification_from_dict(spec_dict, local_data_dict, ...)	Returns item specification from a dictionary.
plugins_dirs(app_settings)	Loads plugins.
load_plugin_dict(plugin_dir, logger)	Loads plugin dict from plugin directory.
load_plugin_specifications(plugin_dict, ...)	Loads plugin's specifications.
load_specification_local_data(config_dir)	Loads specifications' project-specific data.
parameter_identifier(database, parameter, names, ...)	Concatenates given information into parameter value identifier string.
signal_waiter(signal)	
inquire_index_name(model, column, title, parent_widget)	Asks for indexed parameter's index name and updates model accordingly.
preferred_row_height(widget[, factor])	
restore_ui(window, app_settings, settings_group)	Restores UI state from previous session.
save_ui(window, app_settings, settings_group)	Saves UI state for next session.
bisect_chunks(current_data, new_data[, key])	
load_project_dict(project_config_dir, logger)	Loads project dictionary from project directory.
load_local_project_data(project_config_dir, logger)	Loads local project data.
merge_dicts(source, target)	Merges two dictionaries that may contain nested dictionaries recursively.

continues on next page

Table 1 – continued from previous page

fix_lightness_color(color[, lightness])	
scrolling_to_bottom(widget[, tolerance])	
_is_metadata_item(item)	Identifies a database metadata record.
separate_metadata_and_item_metadata(db_map_data)	Separates normal metadata items from item metadata items.

Attributes

_matplotlib_version	
DB_ITEM_SEPARATOR	Display string to separate items such as entity names.

spinetoolbox.helpers.**_matplotlib_version**

class spinetoolbox.helpers.**LinkType**

Bases: enum.Enum

Graphics scene's link types.

CONNECTION = connection

JUMP = jump

spinetoolbox.helpers.**home_dir**()

Returns user's home dir

spinetoolbox.helpers.**format_log_message**(*msg_type*, *message*, *show_datetime=True*)

Adds color tags and optional time stamp to message.

Parameters

- **msg_type** (*str*) – message's type; accepts only 'msg', 'msg_success', 'msg_warning', or 'msg_error'
- **message** (*str*) – message to format
- **show_datetime** (*bool*) – True to add time stamp, False to omit it

Returns
formatted message

Return type
str

spinetoolbox.helpers.**busy_effect**(*func*)

Decorator to change the mouse cursor to 'busy' while a function is processed.

Parameters
func (*Callable*) – Decorated function.

spinetoolbox.helpers.**create_dir**(*base_path*, *folder=''*, *verbosity=False*)

Create (input/output) directories recursively.

Parameters

- **base_path** (*str*) – Absolute path to wanted dir
- **folder** (*str*) – (Optional) Folder name. Usually short name of item.
- **verbosity** (*bool*) – True prints a message that tells if the directory already existed or if it was created.

Raises
OSError if operation failed. –

`spinetoolbox.helpers.rename_dir(old_dir, new_dir, toolbox, box_title)`

Renames directory. Called by `ProjectItemModel.set_item_name()`

Parameters

- **old_dir** (*str*) – Absolute path to directory that will be renamed
- **new_dir** (*str*) – Absolute path to new directory
- **toolbox** (`ToolboxUI`) – A toolbox to log messages and ask questions.
- **box_title** (*str*) – The title of the message boxes, (e.g. "Undoing 'rename DC1 to DC2'")

Returns
True if operation was successful, False otherwise

Return type
bool

`spinetoolbox.helpers.open_url(url)`

Opens the given url in the appropriate Web browser for the user's desktop environment, and returns true if successful; otherwise returns false.

If the URL is a reference to a local file (i.e., the URL scheme is "file") then it will be opened with a suitable application instead of a Web browser.

Handle return value on caller side.

Parameters
url (*str*) – URL to open

Returns
True if successful, False otherwise

Return type
bool

`spinetoolbox.helpers.set_taskbar_icon()`

Set application icon to Windows taskbar.

`spinetoolbox.helpers.supported_img_formats()`

Checks if reading .ico files is supported.

`spinetoolbox.helpers.pyside2_version_check()`

Check that PySide2 version is 5.14 or 5.15. Version 5.15 is allowed but it is not promoted yet because user's may need to update their VC++ runtime libraries on Windows.

qt_version is the Qt version used to compile PySide2 as string. E.g. "5.14.2" qt_version_info is a tuple with each version component of Qt used to compile PySide2. E.g. (5, 14, 2)

`spinetoolbox.helpers.get_datetime(show, date=True)`

Returns date and time string for appending into Event Log messages.

Parameters

- **show** (*bool*) – True returns date and time string. False returns empty string.
- **date** (*bool*) – Whether or not the date should be included in the result

Returns
datetime string or empty string if show is False

Return type
str

`spinetoolbox.helpers.copy_files(src_dir, dst_dir, includes=None, excludes=None)`

Function for copying files. Does not copy folders.

Parameters

- **src_dir** (*str*) – Source directory
- **dst_dir** (*str*) – Destination directory
- **includes** (*list, optional*) – Included files (wildcards accepted)
- **excludes** (*list, optional*) – Excluded files (wildcards accepted)

Returns
Number of files copied

Return type
count (int)

`spinetoolbox.helpers.erase_dir(path, verbosity=False)`

Deletes a directory and all its contents without prompt.

Parameters

- **path** (*str*) – Path to directory
- **verbosity** (*bool*) – Print logging messages or not

Returns
True if operation was successful, False otherwise

Return type
bool

`spinetoolbox.helpers.recursive_overwrite(logger, src, dst, ignore=None, silent=True)`

Copies everything from source directory to destination directory recursively. Overwrites existing files.

Parameters

- **logger** (LoggerInterface) – Enables e.g. printing to Event Log
- **src** (*str*) – Source directory
- **dst** (*str*) – Destination directory
- **ignore** (*Callable, optional*) – Ignore function
- **silent** (*bool*) – If False, messages are sent to Event Log, If True, copying is done in silence

`spinetoolbox.helpers.tuple_itemgetter(itemgetter_func, num_indexes)`

Change output of itemgetter to always be a tuple even for a single index.

Parameters

- **itemgetter_func** (*Callable*) – item getter function
- **num_indexes** (*int*) – number of indexes

Returns
getter function that works with a single index

Return type
Callable

spinetoolbox.helpers.**format_string_list**(*str_list*)

Returns a html unordered list from the given list of strings. Intended to print error logs as returned by spinedb_api.

Parameters
str_list (*list of str*) – list of strings to format

Returns
formatted list

Return type
str

spinetoolbox.helpers.**rows_to_row_count_tuples**(*rows*)

Breaks a list of rows into a list of (row, count) tuples to corresponding chunks of successive rows.

Parameters
rows (*Iterable*) – rows

Returns
row count tuples

Return type
list of tuple

class spinetoolbox.helpers.**IconListManager**(*icon_size*)

A class to manage icons for icon list widgets.

Parameters
icon_size (*QSize*) – icon's size

init_model()

Init model that can be used to display all icons in a list.

_model_data(*index*, *role*)

Creates pixmaps as they're requested by the data() method, to reduce loading time.

Parameters

- **index** (*QModelIndex*) – index to the model
- **role** (*int*) – data role

Returns
role-dependent model data

Return type
Any

spinetoolbox.helpers.**object_icon**(*display_icon*)

Creates and returns a QIcon corresponding to display_icon.

Parameters
display_icon (*int*) – icon id

Returns
requested icon

Return type
QIcon

class spinetoolbox.helpers.**TransparentIconEngine**

Bases: PySide2.QtGui.QIconEngine

Specialization of QIconEngine with transparent background.

pixmap(*size=QSize(512, 512)*, *mode=None*, *state=None*)

class spinetoolbox.helpers.**CharIconEngine**(*char*, *color=None*)

Bases: *TransparentIconEngine*

Specialization of QIconEngine used to draw font-based icons.

Parameters

- **char** (*str*) – character to use as the icon
- **color** (*QColor, optional*) –

paint(*painter*, *rect*, *mode=None*, *state=None*)

class spinetoolbox.helpers.**ColoredIcon**(*icon_file_name*, *icon_color*, *icon_size*, *colored=None*)

Bases: PySide2.QtGui.QIcon

set_colored(*colored*)

color(*mode=QIcon.Normal*)

class spinetoolbox.helpers.**ColoredIconEngine**(*icon_file_name*, *icon_color*, *icon_size*, *colored=None*)

Bases: PySide2.QtGui.QIconEngine

color(*mode=QIcon.Normal*)

set_colored(*colored*)

_do_make_pixmap(*mode*, *state*)

_make_pixmap(*mode*, *state*)

pixmap(*size*, *mode*, *state*)

spinetoolbox.helpers.**color_pixmap**(*pixmap*, *color*)

spinetoolbox.helpers.**make_icon_id**(*icon_code*, *color_code*)

Takes icon and color codes, and return equivalent integer.

Parameters

- **icon_code** (*int*) – icon's code
- **color_code** (*int*) – color code

Returns
icon id

Return type
int

spinetoolbox.helpers.**interpret_icon_id**(*display_icon*)

Takes a display icon id and returns an equivalent tuple of icon and color code.

Parameters
display_icon (*int, optional*) – icon id

Returns
icon's code, color code

Return type
tuple

spinetoolbox.helpers.**default_icon_id**()

Creates a default icon id.

Returns
default icon's id

Return type
int

class spinetoolbox.helpers.**ProjectDirectoryIconProvider**

Bases: PySide2.QtWidgets.QFileIconProvider

QFileIconProvider that provides a Spine icon to the Open Project Dialog when a Spine Toolbox project directory is encountered.

icon(*info*)

Returns an icon for the file described by info.

Parameters
info (*QFileInfo*) – File (or directory) info

Returns
Icon for a file system resource with the given info

Return type
QIcon

spinetoolbox.helpers.**ensure_window_is_on_screen**(*window*, *size*)

Checks if window is on screen and if not, moves and resizes it to make it visible on the primary screen.

Parameters

- **window** (*QWidget*) – a window to check
- **size** (*QSize*) – desired window size if the window is moved

spinetoolbox.helpers.**first_non_null**(*s*)

Returns the first element in Iterable s that is not None.

spinetoolbox.helpers.**get_save_file_name_in_last_dir**(*qsettings*, *key*, *parent*, *caption*, *given_dir*, *filter_=''*)

Calls QFileDialog.getSaveFileName in the directory that was selected last time the dialog was accepted.

Parameters

- **qsettings** (*QSettings*) – A QSettings object where the last directory is stored
- **key** (*string*) – The name of the entry in the above QSettings
- **parent** – Args passed to QFileDialog.getSaveFileName
- **caption** – Args passed to QFileDialog.getSaveFileName

- **given_dir** – Args passed to QFileDialog.getSaveFileName
- **filter** – Args passed to QFileDialog.getSaveFileName

Returns
filename str: selected filter

Return type
str

spinetoolbox.helpers.**get_open_file_name_in_last_dir**(*qsettings*, *key*, *parent*, *caption*, *given_dir*, *filter_=''*)

spinetoolbox.helpers.**try_number_from_string**(*text*)

Tries to convert a string to integer or float.

Parameters
text (*str*) – string to convert

Returns
converted value or text if conversion failed

Return type
int or float or str

spinetoolbox.helpers.**focused_widget_has_callable**(*parent*, *callable_name*)

Returns True if the currently focused widget or one of its ancestors has the given callable.

spinetoolbox.helpers.**call_on_focused_widget**(*parent*, *callable_name*)

Calls the given callable on the currently focused widget or one of its ancestors.

class spinetoolbox.helpers.**ChildCyclingKeyPressFilter**

Bases: PySide2.QtCore.QObject

Event filter class for catching next and previous child key presses. Used in filtering the Ctrl+Tab and Ctrl+Shift+Tab key presses in the Item Properties tab widget.

eventFilter(*obj*, *event*)

spinetoolbox.helpers.**select_gams_executable**(*parent*, *line_edit*)

Opens file browser where user can select a Gams executable (i.e. gams.exe on Windows).

Parameters

- **parent** (*QWidget, optional*) – Parent widget for the file dialog and message boxes
- **line_edit** (*QLineEdit*) – Line edit where the selected path will be inserted

spinetoolbox.helpers.**select_julia_executable**(*parent*, *line_edit*)

Opens file browser where user can select a Julia executable (i.e. julia.exe on Windows). Used in SettingsWidget and KernelEditor.

Parameters

- **parent** (*QWidget, optional*) – Parent widget for the file dialog and message boxes
- **line_edit** (*QLineEdit*) – Line edit where the selected path will be inserted

spinetoolbox.helpers.**select_julia_project**(*parent*, *line_edit*)

Shows file browser and inserts selected julia project dir to give line_edit. Used in SettingsWidget and KernelEditor.

Parameters

- **parent** (*QWidget, optional*) – Parent of QFileDialog
- **line_edit** (*QLineEdit*) – Line edit where the selected path will be inserted

spinetoolbox.helpers.**select_python_interpreter**(*parent*, *line_edit*)

Opens file browser where user can select a python interpreter (i.e. python.exe on Windows). Used in SettingsWidget and KernelEditor.

Parameters

- **parent** (*QWidget*) – Parent widget for the file dialog and message boxes
- **line_edit** (*QLineEdit*) – Line edit where the selected path will be inserted

spinetoolbox.helpers.**select_conda_executable**(*parent*, *line_edit*)

Opens file browser where user can select a conda executable.

Parameters

- **parent** (*QWidget*) – Parent widget for the file dialog and message boxes
- **line_edit** (*QLineEdit*) – Line edit where the selected path will be inserted

spinetoolbox.helpers.**select_certificate_directory**(*parent*, *line_edit*)

Shows file browser and inserts selected certificate directory to given line edit.

Parameters

- **parent** (*QWidget, optional*) – Parent of QFileDialog
- **line_edit** (*QLineEdit*) – Line edit where the selected dir path will be inserted

spinetoolbox.helpers.**file_is_valid**(*parent*, *file_path*, *msgbox_title*, *extra_check=None*)

Checks that given path is not a directory and it's a file that actually exists. In addition, can be used to check if the file name in given file path starts with the given extra_check string. Needed in SettingsWidget and KernelEditor because the QLineEdits are editable. Returns True when file_path is an empty string so that we can use default values (e.g. from line edit place holder text). Returns also True when file_path is just 'python' or 'julia' so that user's can use the python or julia in PATH.

Parameters

- **parent** (*QWidget*) – Parent widget for the message boxes
- **file_path** (*str*) – Path to check
- **msgbox_title** (*str*) – Title for message boxes
- **extra_check** (*str, optional*) – String that must match the file name of the given file_path (without extension)

Returns

True if given path is an empty string or if path is valid, False otherwise

Return type

bool

spinetoolbox.helpers.**dir_is_valid**(*parent*, *dir_path*, *msgbox_title*)

Checks that given path is a directory. Needed in SettingsWdiget and KernelEditor because the QLineEdits are editable. Returns True when dir_path is an empty string so that we can use default values (e.g. from line edit place holder text)

Parameters

- **parent** (*QWidget*) – Parent widget for the message box

- **dir_path** (*str*) – Directory path to check
- **msgbox_title** (*str*) – Message box title

Returns
True if given path is an empty string or if path is an existing directory, False otherwise

Return type
bool

class spinetoolbox.helpers.**QuietLogger**

__getattr__(_)

__call__(**args*, ***kwargs*)

spinetoolbox.helpers.**make_settings_dict_for_engine**(*app_settings*)

Converts Toolbox settings to a dictionary acceptable by Engine.

Parameters
app_settings (*QSettings*) – Toolbox settings

Returns
Engine-compatible settings

Return type
dict

spinetoolbox.helpers.**make_icon_background**(*color*)

spinetoolbox.helpers.**make_icon_toolbar_ss**(*color*)

spinetoolbox.helpers.**color_from_index**(*i*, *count*, *base_hue=0.0*, *saturation=1.0*)

spinetoolbox.helpers.**unique_name**(*prefix*, *existing*)

Creates a unique name in the form "prefix X" where X is a number.

Parameters

- **prefix** (*str*) – name prefix
- **existing** (*Iterable of str*) – existing names

Returns
unique name

Return type
str

spinetoolbox.helpers.**get_upgrade_db_promt_text**(*url*, *current*, *expected*)

spinetoolbox.helpers.**parse_specification_file**(*spec_path*, *logger*)

Parses specification file.

Parameters

- **spec_path** (*str*) – path to specification file
- **logger** (LoggerInterface) – a logger

Returns
specification dict or None if the operation failed

Return type
dict

`spinetoolbox.helpers.load_specification_from_file(spec_path, local_data_dict, spec_factories, app_settings, logger)`

Returns an Item specification from a definition file.

Parameters

- **spec_path** (*str*) – Path of the specification definition file
- **local_data_dict** (*dict*) – specifications local data dict
- **spec_factories** (*dict*) – Dictionary mapping specification type to ProjectItemSpecificationFactory
- **app_settings** (*QSettings*) – Toolbox settings
- **logger** (LoggerInterface) – a logger

Returns
item specification or None if reading the file failed

Return type
ProjectItemSpecification

`spinetoolbox.helpers.specification_from_dict(spec_dict, local_data_dict, spec_factories, app_settings, logger)`

Returns item specification from a dictionary.

Parameters

- **spec_dict** (*dict*) – Dictionary with the specification
- **local_data_dict** (*dict*) – specifications local data
- **spec_factories** (*dict*) – Dictionary mapping specification name to ProjectItemSpecificationFactory
- **app_settings** (*QSettings*) – Toolbox settings
- **logger** (LoggerInterface) – a logger

Returns
specification or None if factory isn't found.

Return type
ProjectItemSpecification or NoneType

`spinetoolbox.helpers.plugins_dirs(app_settings)`

Loads plugins.

Parameters
app_settings (*QSettings*) – Toolbox settings

Returns
plugin directories

Return type
list of str

`spinetoolbox.helpers.load_plugin_dict(plugin_dir, logger)`

Loads plugin dict from plugin directory.

Parameters

- **plugin_dir** (*str*) – path of plugin dir with "plugin.json" in it

- **logger** (LoggerInterface) – a logger

Returns
plugin dict or None if the operation failed

Return type
dict

spinetoolbox.helpers.**load_plugin_specifications**(*plugin_dict*, *local_data_dict*, *spec_factories*, *app_settings*, *logger*)

Loads plugin's specifications.

Parameters

- **plugin_dict** (*dict*) – plugin dict
- **local_data_dict** (*dict*) – specifications local data dictionary
- **spec_factories** (*dict*) – Dictionary mapping specification name to ProjectItemSpecificationFactory
- **app_settings** (*QSettings*) – Toolbox settings
- **logger** (LoggerInterface) – a logger

Returns
mapping from plugin name to list of specifications or None if the operation failed

Return type
dict

spinetoolbox.helpers.**load_specification_local_data**(*config_dir*)

Loads specifications' project-specific data.

Parameters
config_dir (*str or Path*) – project config dir

Returns
specifications local data

Return type
dict

spinetoolbox.helpers.**DB_ITEM_SEPARATOR =**

Display string to separate items such as entity names.

spinetoolbox.helpers.**parameter_identifier**(*database*, *parameter*, *names*, *alternative*)

Concatenates given information into parameter value identifier string.

Parameters

- **database** (*str, optional*) – database's code name
- **parameter** (*str*) – parameter's name
- **names** (*list of str*) – name of the entity or class that holds the value
- **alternative** (*str or NoneType*) – name of the value's alternative

class spinetoolbox.helpers.**SignalWaiter**

Bases: PySide2.QtCore.QObject

A 'traffic light' that allows waiting for a signal to be emitted in another thread.

trigger(**args*)

Signal receiving slot.

wait()

Wait for signal to be received.

spinetoolbox.helpers.**signal_waiter**(*signal*)

class spinetoolbox.helpers.**CustomSyntaxHighlighter**(**arg*, ***kwargs*)

Bases: PySide2.QtGui.QSyntaxHighlighter

property formats

set_style(*style*)

yield_formats(*text*)

highlightBlock(*text*)

spinetoolbox.helpers.**inquire_index_name**(*model*, *column*, *title*, *parent_widget*)

Asks for indexed parameter's index name and updates model accordingly.

Parameters

- **model** (*IndexedValueTableModel or ArrayModel*) – a model with header that contains index names
- **column** (*int*) – column index
- **title** (*str*) – input dialog's title
- **parent_widget** (*QWidget*) – dialog's parent widget

spinetoolbox.helpers.**preferred_row_height**(*widget*, *factor=1.5*)

spinetoolbox.helpers.**restore_ui**(*window*, *app_settings*, *settings_group*)

Restores UI state from previous session.

Parameters

- **window** (*QMainWindow*) –
- **app_settings** (*QSettings*) –
- **settings_group** (*str*) –

spinetoolbox.helpers.**save_ui**(*window*, *app_settings*, *settings_group*)

Saves UI state for next session.

Parameters

- **window** (*QMainWindow*) –
- **app_settings** (*QSettings*) –
- **settings_group** (*str*) –

spinetoolbox.helpers.**bisect_chunks**(*current_data*, *new_data*, *key=None*)

class spinetoolbox.helpers.**FetchParent**

class Init

Bases: `enum.Enum`

Generic enumeration.

Derive from this class to define new enumerations.

UNINITIALIZED

IN_PROGRESS

FINISHED

FAILED

abstract property fetch_item_type

Returns the type of item to fetch, e.g., "object_class". Used to create an initial query for this item.

Returns
str

property is_fetched

property is_busy_fetching

fetch_token

_is_fetched = False

_is_busy_fetching = False

query

query_key

query_iterator

query_initialized

filter_query(*query*, *subquery*, *db_map*)

Filters the initial query created using the `fetch_item_type` property.

Parameters

- **query** (*Query*) – The query
- **subquery** (*Alias*) – The source of the query
- **db_map** (*DiffDatabaseMapping*) –

Returns
Query

fetch_status_change()

Called when fetch status changes.

set_fetched(*fetched*)

Sets the fetched status.

Parameters
fetched (*bool*) – whether parent has been fetched completely

set_busy_fetching(*busy*)

Sets the busy status.

Parameters
busy (*bool*) – whether the parent is busy

reset_fetching(*fetch_token*)

Resets fetch parent as if nothing was ever fetched.

Parameters
fetch_token (*object*) – current fetch token

class spinetoolbox.helpers.**ItemTypeFetchParent**(*fetch_item_type*)

Bases: *FetchParent*

property fetch_item_type

Returns the type of item to fetch, e.g., "object_class". Used to create an initial query for this item.

Returns
str

spinetoolbox.helpers.**load_project_dict**(*project_config_dir*, *logger*)

Loads project dictionary from project directory.

Parameters

- **project_config_dir** (*str*) – project's .spinetoolbox directory
- **logger** (*LoggerInterface*) – a logger

Returns
project dictionary

Return type
dict

spinetoolbox.helpers.**load_local_project_data**(*project_config_dir*, *logger*)

Loads local project data.

Parameters

- **project_config_dir** (*Path or str*) – project's .spinetoolbox directory
- **logger** (*LoggerInterface*) – a logger

Returns
project's local data

Return type
dict

spinetoolbox.helpers.**merge_dicts**(*source*, *target*)

Merges two dictionaries that may contain nested dictionaries recursively.

Parameters

- **source** (*dict*) – dictionary that will be merged to `target`
- **target** (*dict*) – target dictionary

spinetoolbox.helpers.**fix_lightness_color**(*color*, *lightness=240*)

spinetoolbox.helpers.**scrolling_to_bottom**(*widget*, *tolerance=1*)

`spinetoolbox.helpers.`**`_is_metadata_item`**(*item*)

Identifies a database metadata record.

Parameters
item (*dict*) – database item

Returns
True if item is metadata item, False otherwise

Return type
bool

`spinetoolbox.helpers.`**`separate_metadata_and_item_metadata`**(*db_map_data*)

Separates normal metadata items from item metadata items.

Parameters
db_map_data (*dict*) – database records

Returns
item metadata records and metadata records

Return type
tuple

class `spinetoolbox.helpers.`**`HTMLTagFilter`**

Bases: `html.parser.HTMLParser`

HTML tag filter.

Initialize and reset this instance.

If convert_charrefs is True (the default), all character references are automatically converted to the corresponding Unicode characters.

`drain`()

`handle_data`(*data*)

`handle_starttag`(*tag*, *attrs*)

spinetoolbox.link

Classes for drawing graphics items on QGraphicsScene.

authors

M. Marin (KTH), P. Savolainen (VTT)

date
4.4.2018

Module Contents

Classes

LinkBase	Base class for Link and LinkDrawer.
_IconBase	Base class for icons to show over a Link.
_SvgIcon	A svg icon to show over a Link.
_TextIcon	A font awesome icon to show over a Link.
JumpOrLink	Base class for Jump and Link.
Link	A graphics item to represent the connection between two project items.
JumpLink	A graphics icon to represent a jump connection between items.
LinkDrawerBase	A base class for items intended for drawing links between project items.
ConnectionLinkDrawer	An item for drawing connection links between project items.
JumpLinkDrawer	An item for drawing jump connections between project items.

Functions

_regular_poligon_points(n, side[, initial_angle])	

Attributes

LINK_COLOR	
JUMP_COLOR	

spinetoolbox.link.**LINK_COLOR**

spinetoolbox.link.**JUMP_COLOR**

class spinetoolbox.link.**LinkBase**(*toolbox*, *src_connector*, *dst_connector*)

Bases: PySide2.QtWidgets.QGraphicsPathItem

Base class for Link and LinkDrawer.

Mainly provides the `update_geometry` method for 'drawing' the link on the scene.

Parameters

- **toolbox** (ToolboxUI) – main UI class instance
- **src_connector** (ConnectorButton, *optional*) – Source connector button
- **dst_connector** (ConnectorButton) – Destination connector button

property outline_color

property magic_number

property src_rect

Returns the scene rectangle of the source connector.

property src_center

Returns the center point of the source rectangle.

property dst_rect

Returns the scene rectangle of the destination connector.

property dst_center

Returns the center point of the destination rectangle.

_COLOR

shape()

moveBy(*_dx*, *_dy*)

Does nothing. This item is not moved the regular way, but follows the ConnectorButtons it connects.

update_geometry(*curved_links=None*)

Updates geometry.

guide_path()

For tests.

_do_update_geometry()

Sets the path for this item.

_add_ellipse_path(*path*)

Adds an ellipse for the link's base.

Parameters
QPainterPath –

_get_joint_angle()

_add_arrow_path(*path*)

Returns an arrow path for the link's tip.

Parameters
QPainterPath –

static _get_offset(*button*)

_get_src_offset()

_get_dst_offset()

_find_new_point(*points*, *target*)

Finds a new point that approximates points to target in a smooth trajectory. Returns the new point, or None if no need for approximation.

Parameters

- **points** (*list(QPointF)*) –
- **target** (*QPointF*) –

Returns
QPointF or None

_close_enough(*p1*, *p2*)

_make_guide_path(*curved_links=False*)

Returns a 'narrow' path connecting this item's source and destination.

Parameters
curved_links (*bool*) – Whether the path should follow a curved line or just a straight line

Returns
QPainterPath

itemChange(*change*, *value*)

Wipes out the link when removed from scene.

wipe_out()

Removes any trace of this item from the system.

class spinetoolbox.link.**_IconBase**(*x*, *y*, *w*, *h*, *parent*, *tooltip=None*, *active=True*)

Bases: PySide2.QtWidgets.QGraphicsEllipseItem

Base class for icons to show over a Link.

hoverEnterEvent(*event*)

hoverLeaveEvent(*event*)

class spinetoolbox.link.**_SvgIcon**(*parent*, *extent*, *path*, *tooltip=None*, *active=False*)

Bases: *_IconBase*

A svg icon to show over a Link.

wipe_out()

Cleans up icon's resources.

class spinetoolbox.link.**_TextIcon**(*parent*, *extent*, *char*, *tooltip=None*, *active=False*)

Bases: *_IconBase*

A font awesome icon to show over a Link.

FONT_SIZE_PIXELS = 16

wipe_out()

Cleans up icon's resources.

class spinetoolbox.link.**JumpOrLink**(*toolbox*, *src_connector*, *dst_connector*)

Bases: *LinkBase*

Base class for Jump and Link.

Parameters

- **toolbox** (ToolboxUI) – main UI class instance
- **src_connector** (ConnectorButton, *optional*) – Source connector button
- **dst_connector** (ConnectorButton) – Destination connector button

abstract property item

_do_update_geometry()

See base class.

_place_icons()

mousePressEvent(*e*)

Ignores event if there's a connector button underneath, to allow creation of new links.

Parameters

e (*QGraphicsSceneMouseEvent*) – Mouse event

contextMenuEvent(*e*)

Selects the link and shows context menu.

Parameters

e (*QGraphicsSceneMouseEvent*) – Mouse event

paint(*painter*, *option*, *widget=None*)

Sets a dashed pen if selected.

shape()

wipe_out()

Removes any trace of this item from the system.

_make_execution_animation()

Returns an animation to play when execution 'passes' through this link.

Returns

QVariantAnimation

run_execution_animation()

Runs execution animation.

_handle_execution_animation_value_changed(*step*)

class spinetoolbox.link.**Link**(*toolbox*, *src_connector*, *dst_connector*, *connection*)

Bases: *JumpOrLink*

A graphics item to represent the connection between two project items.

Parameters

- **toolbox** (ToolboxUI) – main UI class instance
- **src_connector** (ConnectorButton) – Source connector button
- **dst_connector** (ConnectorButton) – Destination connector button
- **connection** (LoggingConnection) – connection this link represents

property name

property connection

property item

_COLOR

_MEMORY =

_FILTERS =

_PURGE =

_DATAPACKAGE = :/icons/datapkg.svg

update_icons()

itemChange(*change*, *value*)

Brings selected link to top.

class spinetoolbox.link.**JumpLink**(*toolbox*, *src_connector*, *dst_connector*, *jump*)

Bases: *JumpOrLink*

A graphics icon to represent a jump connection between items.

Parameters

- **toolbox** (ToolboxUI) – main UI class instance
- **src_connector** (ConnectorButton) – Source connector button
- **dst_connector** (ConnectorButton) – Destination connector button
- **jump** (*spine_engine.project_item.connection.Jump*) – connection this link represents

property jump

property item

property name

_COLOR

_ISSUE =

issues()

Checks if jump is well-defined.

Returns
issues regarding the jump

Return type
list of str

update_icons()

class spinetoolbox.link.**LinkDrawerBase**(*toolbox*)

Bases: *LinkBase*

A base class for items intended for drawing links between project items.

Parameters
toolbox (ToolboxUI) – main UI class instance

property src_rect

Returns the scene rectangle of the source connector.

property dst_rect

Returns the scene rectangle of the destination connector.

property dst_center

Returns the center point of the destination rectangle.

_get_dst_offset()

abstract add_link()

Makes link between source and destination connectors.

wake_up(*src_connector*)

Sets the source connector, shows this item and adds it to the scene. After calling this, the scene is in link drawing mode.

Parameters
src_connector (ConnectorButton) – source connector

sleep()

Removes this drawer from the scene, clears its source and destination connectors, and hides it. After calling this, the scene is no longer in link drawing mode.

class spinetoolbox.link.**ConnectionLinkDrawer**(*toolbox*)

Bases: *LinkDrawerBase*

An item for drawing connection links between project items.

Parameters
toolbox (ToolboxUI) – main UI class instance

_COLOR

add_link()

Makes link between source and destination connectors.

wake_up(*src_connector*)

Sets the source connector, shows this item and adds it to the scene. After calling this, the scene is in link drawing mode.

Parameters
src_connector (ConnectorButton) – source connector

sleep()

Removes this drawer from the scene, clears its source and destination connectors, and hides it. After calling this, the scene is no longer in link drawing mode.

class spinetoolbox.link.**JumpLinkDrawer**(*toolbox*)

Bases: *LinkDrawerBase*

An item for drawing jump connections between project items.

Parameters
toolbox (ToolboxUI) – main UI class instance

_COLOR

add_link()

Makes link between source and destination connectors.

spinetoolbox.link.**_regular_poligon_points**(*n*, *side*, *initial_angle=0*)

spinetoolbox.load_project_items

Functions to load project item modules.

author

A. Soininen (VTT)

date

29.4.2020

Module Contents

Functions

load_project_items(items_package_name)	Loads project item modules.
_find_module_material(module)	

spinetoolbox.load_project_items.**load_project_items**(*items_package_name*)

Loads project item modules.

Parameters

items_package_name (*str*) – name of the package that contains the project items

Returns

two dictionaries; first maps item type to its category
while second maps item type to item factory

Return type

tuple of dict

spinetoolbox.load_project_items.**_find_module_material**(*module*)

spinetoolbox.log_mixin

Contains LogMixin.

authors

M. Marin (ER)

date

9.12.2021

Module Contents

Classes

LogMixin

class spinetoolbox.log_mixin.**LogMixin**

add_log_message(*filter_id*, *message*)

Adds a message to the log document.

Parameters

- **filter_id** (*str*) – filter identifier
- **message** (*str*) – formatted message

add_event_message(*filter_id*, *msg_type*, *msg_text*)

Adds a message to the log document.

Parameters

- **filter_id** (*str*) – filter identifier
- **msg_type** (*str*) – message type
- **msg_text** (*str*) – message text

add_process_message(*filter_id*, *msg_type*, *msg_text*)

Adds a message to the log document.

Parameters

- **filter_id** (*str*) – filter identifier
- **msg_type** (*str*) – message type
- **msg_text** (*str*) – message text

spinetoolbox.logger_interface

A logger interface.

authors

A. Soininen (VTT)

date

16.1.2020

Module Contents

Classes

LoggerInterface	Placeholder for signals that can be emitted to send messages to an output device.

class spinetoolbox.logger_interface.**LoggerInterface**

Bases: PySide2.QtCore.QObject

Placeholder for signals that can be emitted to send messages to an output device.

The signals should be connected to a concrete logging system.

Currently, this is just a 'model interface'. ToolboxUI contains the same signals so it can be used as a drop-in replacement for this class.

msg

Emits a notification message.

msg_success

Emits a message on success

msg_warning

Emits a warning message.

msg_error

Emits an error message.

msg_proc

Emits a message originating from a subprocess (usually something printed to stdout).

msg_proc_error

Emits an error message originating from a subprocess (usually something printed to stderr).

information_box

Requests an 'information message box' (e.g. a message window) to be opened with a given title and message.

error_box

Requests an 'error message box' to be opened with a given title and message.

spinetoolbox.main

Provides the main() function.

author

A. Soininen (VTT)

date

4.10.2019

Module Contents

Functions

main()	Creates main window GUI and starts main event loop.
_make_argument_parser()	Returns a command line argument parser configured for Toolbox use.
_add_pywin32_system32_to_path()	Adds a directory to PATH on Windows that is required to make pywin32 work

Attributes

dirname
plugin_path

spinetoolbox.main.**dirname**

spinetoolbox.main.**plugin_path**

spinetoolbox.main.**main**()

Creates main window GUI and starts main event loop.

spinetoolbox.main.**_make_argument_parser**()

Returns a command line argument parser configured for Toolbox use.

Returns
Toolbox' command line argument parser

Return type
ArgumentParser

spinetoolbox.main.**_add_pywin32_system32_to_path**()

Adds a directory to PATH on Windows that is required to make pywin32 work on (Conda) Python 3.8. See https://github.com/Spine-project/Spine-Toolbox/issues/1230.

spinetoolbox.metaobject

MetaObject class.

authors

E. Rinne (VTT), P. Savolainen (VTT)

date
18.12.2017

Module Contents

Classes

MetaObject	Class for an object which has a name, type, and some description.

class spinetoolbox.metaobject.**MetaObject**(*name*, *description*)

Bases: PySide2.QtCore.QObject

Class for an object which has a name, type, and some description.

Parameters

- **name** (*str*) – Object name
- **description** (*str*) – Object description

set_name(*name*)

Set object name and short name. Note: Check conflicts (e.g. name already exists) before calling this method.

Parameters
name (*str*) – New (long) name for this object

set_description(*description*)

Set object description.

Parameters
description (*str*) – Object description

spinetoolbox.plotting

Functions for plotting on PlotWidget.

author

A. Soininen (VTT)

date
9.7.2019

Module Contents

Classes

XYData	Two-dimensional data for plotting.
TreeNode	A labeled node in tree structure.
ParameterTableHeaderSection	Header section info for Database editor's parameter tables.

Functions

convert_indexed_value_to_tree(value)	Converts Maps to tree nodes recursively.
turn_nodes_to_xy_data(root_node[, index_names, indexes])	Constructs plottable data and indexes recursively.
raise_if_not_common_x_labels(data_list)	Raises an exception if data has different x axis labels.
raise_if_incompatible_x(data_list)	Raises an exception if the types of x data don't match.
reduce_indexes(data_list)	Removes redundant indexes from given XYData.
combine_data_with_same_indexes(data_list)	Combines data with same data indexes into the same x axis.
plot_data(data_list[, plot_widget])	Returns a plot widget with plots of the given data.
_make_x_plottable(xs)	Converts x-axis values to something matplotlib can handle.
_make_plot_function(data_list, plot_widget)	Decides plot method and default keyword arguments based on XYData.
_clear_plot(plot_widget)	Removes plots and legend from plot widget.
_limit_string_x_tick_labels(data, plot_widget)	Limits the number of x tick labels in case x-axis consists of strings.
_table_display_row(row)	Calculates a human-readable row number.
plot_parameter_table_selection(model, model_indexes, ...)	Returns a plot widget with plots of the selected indexes.
plot_value_editor_table_selection(model, model_indexes)	Returns a plot widget with plots of the selected indexes.
plot_pivot_table_selection(model, model_indexes[, ...])	Returns a plot widget with plots of the selected indexes.
plot_db_mngr_items(items, db_maps[, plot_widget])	Returns a plot widget with plots of database manager parameter value items.
_has_x_column(model, source_model)	Checks if pivot source model has x column.
_set_default_node(root_node, key, label)	Gets node from the contents of root_node adding a new node if necessary.
_get_parsed_value(model_index, display_row)	Gets parsed value from model.
_pivot_index_names(indexes)	Gathers index names from pivot table.
_pivot_display_row(row, source_model)	Calculates display row for pivot table.
_convert_to_leaf(y)	Converts parameter value to leaf TreeElement.
add_row_to_exception(row, display_row)	Adds row information to PlottingError if it is raised in the with block.
add_array_plot(plot_widget, value)	Adds an array plot to a plot widget.
add_time_series_plot(plot_widget, value)	Adds a time series step plot to a plot widget.

Attributes

_BASE_SETTINGS
_SCATTER_PLOT_SETTINGS
_LINE_PLOT_SETTINGS
_TIME_SERIES_PLOT_SETTINGS

spinetoolbox.plotting.**_BASE_SETTINGS**

spinetoolbox.plotting.**_SCATTER_PLOT_SETTINGS**

spinetoolbox.plotting.**_LINE_PLOT_SETTINGS**

spinetoolbox.plotting.**_TIME_SERIES_PLOT_SETTINGS**

exception spinetoolbox.plotting.**PlottingError**(*message*)

Bases: Exception

An exception signalling failure in plotting.

Parameters
message (*str*) – an error message

property message

the error message.

Type
str

class spinetoolbox.plotting.**XYData**

Two-dimensional data for plotting.

x :List[Union[float, int, str, numpy.datetime64]]

y :List[Union[float, int]]

x_label :str

y_label :str

data_index :List[str]

index_names :List[str]

class spinetoolbox.plotting.**TreeNode**

A labeled node in tree structure.

label :str

content :Dict

class spinetoolbox.plotting.**ParameterTableHeaderSection**

Header section info for Database editor's parameter tables.

label :str

separator :Optional[str]

spinetoolbox.plotting.**convert_indexed_value_to_tree**(*value*)

Converts Maps to tree nodes recursively.

Parameters
value (*IndexedValue*) – value to convert

Returns
root node of the converted tree

Return type
TreeNode

Raises
ValueError – raised when leaf value couldn't be converted to float

spinetoolbox.plotting.**turn_nodes_to_xy_data**(*root_node*, *index_names=None*, *indexes=None*)

Constructs plottable data and indexes recursively.

Parameters

- **root_node** (TreeNode) – root node
- **index_names** (*list of str, optional*) – list of current index names
- **indexes** (*list*) – list of current indexes

Yields
XYData – plot data

spinetoolbox.plotting.**raise_if_not_common_x_labels**(*data_list*)

Raises an exception if data has different x axis labels.

Parameters
data_list (*list of* XYData) – data to check

Raises
PlottingError – raised if x axis labels don't match.

spinetoolbox.plotting.**raise_if_incompatible_x**(*data_list*)

Raises an exception if the types of x data don't match.

Parameters
data_list (*list of* XYData) – data to check

Raises
PlottingError – raised if x data types don't match.

spinetoolbox.plotting.**reduce_indexes**(*data_list*)

Removes redundant indexes from given XYData.

Parameters
data_list (*list of* XYData) – data to reduce

Returns
reduced data list and list of common data indexes

Return type
tuple

spinetoolbox.plotting.**combine_data_with_same_indexes**(*data_list*)

Combines data with same data indexes into the same x axis.

Parameters
data_list (*list of* XYData) – data to combine

Returns
combined data

Return type
list of *XYData*

spinetoolbox.plotting.**plot_data**(*data_list*, *plot_widget=None*)

Returns a plot widget with plots of the given data.

Parameters

- **data_list** (*list of* XYData) – data to plot
- **plot_widget** (PlotWidget, *optional*) – an existing plot widget to draw into or None to create a new widget

Returns
a PlotWidget object

spinetoolbox.plotting.**_make_x_plottable**(*xs*)

Converts x-axis values to something matplotlib can handle.

Parameters
xs (*list*) – x values

Returns
x values

Return type
list

spinetoolbox.plotting.**_make_plot_function**(*data_list*, *plot_widget*)

Decides plot method and default keyword arguments based on XYData.

Parameters
data_list (*list of* XYData) – data to plot

Returns
plot method

Return type
Callable

spinetoolbox.plotting.**_clear_plot**(*plot_widget*)

Removes plots and legend from plot widget.

Parameters
plot_widget (PlotWidget) – plot widget

spinetoolbox.plotting.**_limit_string_x_tick_labels**(*data*, *plot_widget*)

Limits the number of x tick labels in case x-axis consists of strings.

Matplotlib tries to plot every single x tick label if they are strings. This can become very slow if the labels are numerous.

Parameters

- **data** (*list of* XYData) – plot data
- **plot_widget** (PlotWidget) – plot widget

spinetoolbox.plotting.**_table_display_row**(*row*)

Calculates a human-readable row number.

Parameters
row (*int*) – model row

Returns
row number

Return type
int

spinetoolbox.plotting.**plot_parameter_table_selection**(*model*, *model_indexes*, *table_header_sections*, *value_section_label*, *plot_widget=None*)

Returns a plot widget with plots of the selected indexes.

Parameters

- **model** (*QAbstractTableModel*) – a model
- **model_indexes** (*Iterable of QModelIndex*) – a list of QModelIndex objects for plotting
- **table_header_sections** (*list of ParameterTableHeaderSection*) – table header labels
- **value_section_label** (*str*) – value column's header label
- **plot_widget** (*PlotWidget, optional*) – an existing plot widget to draw into or None to create a new widget

Returns
a PlotWidget object

Return type
PlotWidget

spinetoolbox.plotting.**plot_value_editor_table_selection**(*model*, *model_indexes*, *plot_widget=None*)

Returns a plot widget with plots of the selected indexes.

Parameters

- **model** (*QAbstractTableModel*) – a model
- **model_indexes** (*Iterable of QModelIndex*) – a list of QModelIndex objects for plotting
- **plot_widget** (*PlotWidget, optional*) – an existing plot widget to draw into or None to create a new widget

Returns
a PlotWidget object

Return type
PlotWidget

spinetoolbox.plotting.**plot_pivot_table_selection**(*model*, *model_indexes*, *plot_widget=None*)

Returns a plot widget with plots of the selected indexes.

Parameters

- **model** (*QAbstractTableModel*) – a model
- **model_indexes** (*Iterable of QModelIndex*) – a list of QModelIndex objects for plotting
- **plot_widget** (*PlotWidget, optional*) – an existing plot widget to draw into or None to create a new widget

Returns
a PlotWidget object

Return type
PlotWidget

spinetoolbox.plotting.**plot_db_mngr_items**(*items*, *db_maps*, *plot_widget=None*)

Returns a plot widget with plots of database manager parameter value items.

Parameters

- **items** (*list of dict*) – parameter value items
- **db_maps** (*list of DatabaseMappingBase*) – database mappings corresponding to items
- **plot_widget** (PlotWidget, *optional*) – widget to add plots to

spinetoolbox.plotting.**_has_x_column**(*model*, *source_model*)

Checks if pivot source model has x column.

Parameters

- **model** (PivotTableSortFilterProxy) – proxy pivot model
- **source_model** (PivotTableModelBase) – pivot table model

Returns
True if x pivot table has column, False otherwise

Return type
bool

spinetoolbox.plotting.**_set_default_node**(*root_node*, *key*, *label*)

Gets node from the contents of root_node adding a new node if necessary.

Parameters

- **root_node** (TreeNode) – root node
- **key** (*Hashable*) – key to root_node contents
- **label** (*str*) – label of possible new node

Returns
node at given key

Return type
TreeNode

spinetoolbox.plotting.**_get_parsed_value**(*model_index*, *display_row*)

Gets parsed value from model.

Parameters

- **model_index** (*QModelIndex*) – model index
- **display_row** (*Callable*) – callable that returns a display row

Returns
parsed value

Return type
Any

Raises
PlottingError – raised if parsing of value failed

spinetoolbox.plotting.**_pivot_index_names**(*indexes*)

Gathers index names from pivot table.

Parameters
indexes (*tuple of str*) – "path" of indexes

Returns
names corresponding to given indexes

Return type
tuple of str

`spinetoolbox.plotting._pivot_display_row(row, source_model)`

Calculates display row for pivot table.

Parameters

- **row** (*int*) – row in source table model
- **source_model** (*QAbstractItemModel*) – pivot model

Returns
human-readable row number

Return type
int

`spinetoolbox.plotting._convert_to_leaf(y)`

Converts parameter value to leaf TreeElement.

Parameters
y (*Any*) – parameter value

Returns
leaf element

Return type
float or datetime or *TreeNode*

`spinetoolbox.plotting.add_row_to_exception(row, display_row)`

Adds row information to PlottingError if it is raised in the with block.

Parameters

- **row** (*int*) – row
- **display_row** (*Callable*) – function to convert row to display row

`spinetoolbox.plotting.add_array_plot(plot_widget, value)`

Adds an array plot to a plot widget.

Parameters

- **plot_widget** (PlotWidget) – a plot widget to modify
- **value** (*Array*) – the array to plot

`spinetoolbox.plotting.add_time_series_plot(plot_widget, value)`

Adds a time series step plot to a plot widget.

Parameters

- **plot_widget** (PlotWidget) – a plot widget to modify
- **value** (*TimeSeries*) – the time series to plot

spinetoolbox.plugin_manager

Contains PluginManager class.

author

M. Marin (KTH)

date

21.2.2021

Module Contents

Classes

PluginManager	Class for managing plugins.
_PluginWorker	

Functions

_download_file(remote, local)
_download_plugin(plugin, plugin_local_dir)

spinetoolbox.plugin_manager.**_download_file**(*remote*, *local*)

spinetoolbox.plugin_manager.**_download_plugin**(*plugin*, *plugin_local_dir*)

class spinetoolbox.plugin_manager.**PluginManager**(*toolbox*)

Class for managing plugins.

Parameters

toolbox (ToolboxUI) – Toolbox instance.

property plugin_toolbars

property plugin_specs

load_installed_plugins()

Loads installed plugins and adds their specifications to toolbars.

reload_plugins_with_local_data()

Reloads plugins that have project specific local data.

load_individual_plugin(*plugin_dir*, *specification_local_data*)

Loads plugin from directory.

Parameters

- **plugin_dir** (*str*) – path of plugin dir with "plugin.json" in it.
- **specification_local_data** (*dict*) – specification local data

_create_worker()

_clean_up_worker(*worker*)

_load_registry()

show_install_plugin_dialog(*_=False*)

_do_show_install_plugin_dialog()

_install_plugin(*plugin_name*)

Installs plugin from the registry and loads it.

Parameters
plugin_name (*str*) – plugin name

_load_installed_plugin(*plugin_local_dir*)

show_manage_plugins_dialog(*_=False*)

_do_show_manage_plugins_dialog()

_remove_plugin(*plugin_name*)

Removes installed plugin.

Parameters
plugin_name (*str*) – plugin name

_update_plugin(*plugin_name*)

exception spinetoolbox.plugin_manager.**PluginWorkFailed**

Bases: Exception

Exception to signal plugin worker that something failed.

Initialize self. See help(type(self)) for accurate signature.

class spinetoolbox.plugin_manager.**_PluginWorker**

Bases: PySide2.QtCore.QObject

failed

finished

succeeded

start(*function*, **args*, ***kwargs*)

_do_work()

clean_up()

spinetoolbox.project

Spine Toolbox project class.

authors

P. Savolainen (VTT), E. Rinne (VTT)

date

10.1.2018

Module Contents

Classes

ItemNameStatus	Generic enumeration.
SpineToolboxProject	Class for Spine Toolbox projects.

Functions

node_successors(g)	Returns a dict mapping nodes in topological order to a list of successors.
_edges_causing_loops(g)	Returns a list of edges whose removal from g results in it becoming acyclic.
_ranks(node_successors)	Calculates node ranks.

class spinetoolbox.project.**ItemNameStatus**

Bases: enum.Enum

Generic enumeration.

Derive from this class to define new enumerations.

OK

INVALID

EXISTS

SHORT_NAME_EXISTS

class spinetoolbox.project.**SpineToolboxProject**(*toolbox*, *name*, *description*, *p_dir*, *plugin_specs*, *settings*, *logger*)

Bases: *spinetoolbox.metaobject.MetaObject*

Class for Spine Toolbox projects.

Parameters

- **toolbox** (ToolboxUI) – toolbox of this project
- **name** (*str*) – Project name
- **description** (*str*) – Project description
- **p_dir** (*str*) – Project directory

- **plugin_specs** (*Iterable of ProjectItemSpecification*) – specifications available as plugins
- **settings** (*QSettings*) – Toolbox settings
- **logger** (LoggerInterface) – a logger instance

property connections

property settings

renamed

Emitted after project has been renamed.

project_about_to_be_torn_down

Emitted before project is being torn down.

project_execution_about_to_start

Emitted just before the entire project is executed.

project_execution_finished

Emitted after the entire project execution finishes.

connection_established

Emitted after new connection has been added to project.

connection_about_to_be_removed

Emitted before connection removal.

connection_updated

Emitted after a connection has been updated.

jump_added

Emitted after a jump has been added.

jump_about_to_be_removed

Emitted before a jump is removed.

jump_updated

Emitted after a jump has been replaced by another.

item_added

Emitted after a project item has been added.

item_about_to_be_removed

Emitted before project item removal.

item_renamed

Emitted after project item has been renamed.

specification_added

Emitted after a specification has been added.

specification_about_to_be_removed

Emitted before a specification will be removed.

specification_replaced

Emitted after a specification has been replaced.

specification_saved

Emitted after a specification has been saved.

toolbox()

Returns Toolbox main window.

Returns
main window

Return type
ToolboxUI

_create_project_structure(*directory*)

Makes the given directory a Spine Toolbox project directory. Creates directories and files that are common to all projects.

Parameters
directory (*str*) – Abs. path to a directory that should be made into a project directory

Returns
True if project structure was created successfully, False otherwise

Return type
bool

call_set_name_and_description(*name*, *description*)

set_name(*name*)

Changes project name.

Parameters
name (*str*) – New project name

set_description(*description*)

Set object description.

Parameters
description (*str*) – Object description

save()

Collects project information and objects into a dictionary and writes it to a JSON file.

_save_all_specifications(*local_path*)

Writes all specifications except plugins to disk.

Local specification data is also written to disk, including local data from plugins.

Parameters
local_path (*Path*) –

Returns
specification local data that is supposed to be stored in a project specific place

Return type
dict

_pop_local_data_from_items_dict(*items_dict*)

Pops local data from project items dict.

Parameters
items_dict (*dict*) – items dict

Returns
local project item data

Return type
dict

static _dump(*target_dict*, *out_stream*)

Dumps given dict into output stream.

Parameters

- **target_dict** (*dict*) – dictionary to dump
- **out_stream** (*IOBase*) – output stream

load(*spec_factories*, *item_factories*)

Loads project from its project directory.

Parameters

- **spec_factories** (*dict*) – Dictionary mapping specification name to ProjectItemSpecificationFactory
- **item_factories** (*dict*) – mapping from item type to ProjectItemFactory

Returns
True if the operation was successful, False otherwise

Return type
bool

static _merge_local_data_to_project_info(*local_data_dict*, *project_info*)

Merges local data into project info.

Parameters

- **local_data_dict** (*dict*) – local data
- **project_info** (*dict*) – project dict

connection_from_dict(*connection_dict*)

jump_from_dict(*jump_dict*)

add_specification(*specification*, *save_to_disk=True*)

Adds a specification to the project.

Parameters

- **specification** (*ProjectItemSpecification*) – specification to add
- **save_to_disk** (*bool*) – if True, save the specification to disk

Returns
A unique identifier for the specification or None if the operation was unsuccessful

Return type
int

is_specification_name_reserved(*name*)

Checks if specification exists.

Parameters
name (*str*) – specification's name

Returns
True if project has given specification, False otherwise

Return type
bool

`specifications()`

Yields project's specifications.

Yields
ProjectItemSpecification – specification

`_specification_id()`

Creates an id for specification.

Returns
new id

Return type
int

`get_specification(name_or_id)`

Returns project item specification.

Parameters
name_or_id (*str or int*) – specification's name or id

Returns
specification or None if specification was not found

Return type
ProjectItemSpecification

`specification_name_to_id(name)`

Returns identifier for named specification.

Parameters
name (*str*) – specification's name

Returns
specification's id or None if no such specification exists

Return type
int

`remove_specification(id_or_name)`

Removes a specification from project.

Parameters
id_or_name (*int or str*) – specification's id or name

`replace_specification(name, specification, save_to_disk=True)`

Replaces an existing specification.

Refreshes the spec in all items that use it.

Parameters

- **name** (*str*) – name of the specification to replace
- **specification** (*ProjectItemSpecification*) – a specification
- **save_to_disk** (*bool*) – If True, saves the given specification to disk

Returns
True if operation was successful, False otherwise

Return type
bool

`save_specification_file`(*specification*, *previous_name=None*)

Saves the given project item specification.

Save path is determined by specification directory and specification's name.

Parameters

- **specification** (*ProjectItemSpecification*) – specification to save
- **previous_name** (*str, optional*) – specification's earlier name if it has been renamed/replaced

Returns
True if operation was successful, False otherwise

Return type
bool

`_update_specification_local_data_store`(*specification*, *local_data*, *previous_name*)

Updates the file containing local data of project's specifications.

Parameters

- **specification** (*ProjectItemSpecification*) – specification
- **local_data** (*dict*) – local data serialized into dict
- **previous_name** (*str, optional*) – specification's earlier name if it has been renamed/replaced

`_default_specification_file_path`(*specification*)

Determines a path inside project directory to save a specification.

Parameters
specification (*ProjectItemSpecification*) – specification

Returns
valid path or None if operation failed

Return type
str

`add_item`(*item*, *silent=True*)

Adds a project to item project.

Parameters

- **item** (ProjectItem) – item to add
- **silent** (*bool*) – if True, don't log messages

`has_items`()

Returns True if project has project items.

Returns
True if project has items, False otherwise

Return type
bool

get_item(*name*)

Returns project item.

Parameters
name (*str*) – item's name

Returns
project item

Return type
ProjectItem

get_items()

Returns all project items.

Returns
all project items

Return type
list of *ProjectItem*

rename_item(*previous_name*, *new_name*, *rename_data_dir_message*)

Renames a project item

Parameters

- **previous_name** (*str*) – item's current name
- **new_name** (*str*) – item's new name
- **rename_data_dir_message** (*str*) – message to show when renaming item's data directory

Returns
True if item was renamed successfully, False otherwise

Return type
bool

validate_project_item_name(*name*)

Validates item name.

Parameters
name (*str*) – proposed project item's name

Returns
validation result

Return type
ItemNameStatus

find_connection(*source_name*, *destination_name*)

Searches for a connection between given items.

Parameters

- **source_name** (*str*) – source item's name
- **destination_name** (*str*) – destination item's name

Returns
connection instance or None if there is no connection

Return type
Connection

`connections_for_item`(*item_name*)

Returns connections that have given item as source or destination.

Parameters
item_name (*str*) – item's name

Returns
connections connected to item

Return type
list of Connection

`add_connection`(*connection*, *silent=False*)

Adds a connection to the project.

Parameters

- **connection** (*Connection*) – connection to add
- **silent** (*bool*) – If False, prints 'Link establ…' msg to Event Log

Returns
True if connection was added successfully, False otherwise

Return type
bool

`remove_connection`(*connection*)

Removes a connection from the project.

Parameters
connection (LoggingConnection) – connection to remove

`update_connection`(*connection*, *source_position*, *destination_position*)

Updates existing connection between items.

Updating does not trigger any updates to the DAG or project items.

Parameters

- **connection** (LoggingConnection) – connection to update
- **source_position** (*str*) – link's position on source item's icon
- **destination_position** (*str*) – link's position on destination item's icon

`jumps_for_item`(*item_name*)

Returns jumps that have given item as source or destination.

Parameters
item_name (*str*) – item's name

Returns
jumps connected to item

Return type
list of Jump

`add_jump`(*jump*, *silent=False*)

Adds a jump to project.

Parameters

- **jump** (*Jump*) – jump to add
- **silent** (*bool*) – if True, don't log messages

find_jump(*source_name*, *destination_name*)

Searches for a jump between given items.

Parameters

- **source_name** (*str*) – source item's name
- **destination_name** (*str*) – destination item's name

Returns
connection instance or None if there is no jump

Return type
Jump

remove_jump(*jump*)

Removes a jump from the project.

Parameters
jump (*Jump*) – jump to remove

update_jump(*jump*, *source_position*, *destination_position*)

Updates an existing jump between items.

Parameters

- **jump** (LoggingJump) – jump to update
- **source_position** (*str*) – link's position on source item's icon
- **destination_position** (*str*) – link's position on destination item's icon

_update_jump_icons()

Updates icons for all jumps in the project.

jump_issues(*jump*)

Checks if jump is OK.

Parameters
jump (*Jump*) – jump to check

Returns
list of issues, if any

Return type
list of str

_dag_iterator()

Iterates directed graphs in the project.

Yields
nx.DiGraph

dags()

Used in tests. Returns a list of dags in the project.

Returns
list

node_is_isolated(*node*)

Used in tests. Checks if the project item with the given name has any connections.

Parameters
node (*str*) – Project item name

Returns
bool

dag_with_node(*node*)

restore_project_items(*items_dict*, *item_factories*, *silent*)

Restores project items from dictionary.

Parameters

- **items_dict** (*dict*) – a mapping from item name to item dict
- **item_factories** (*dict*) – a mapping from item type to ProjectItemFactory
- **silent** (*bool*) – if True, suppress a log messages

remove_item_by_name(*item_name*, *delete_data=False*)

Removes project item by its name.

Parameters

- **item_name** (*str*) – Item's name
- **delete_data** (*bool*) – If set to True, deletes the directories and data associated with the item

execute_dags(*dags*, *execution_permits_list*, *msg*)

Executes given dags.

Parameters

- **dags** (*Sequence(DiGraph)*) –
- **execution_permits_list** (*Sequence(dict)*) –
- **msg** (*str*) – message to log before execution

_execute_dags(*dags*, *execution_permits_list*)

create_engine_worker(*dag*, *execution_permits*, *dag_identifier*, *settings*, *job_id*)

Creates and returns a SpineEngineWorker to execute given *validated* dag.

Parameters

- **dag** (*nx.DiGraph*) – The dag
- **execution_permits** (*dict*) – mapping item names to a boolean indicating whether to execute it or skip it
- **dag_identifier** (*str*) – A string identifying the dag, for logging
- **settings** (*dict*) – project and app settings to send to the spine engine.
- **job_id** (*str*) – Job Id for remote execution

Returns
SpineEngineWorker

_handle_engine_worker_finished(*worker*)

execute_selected(*names*)

Executes DAGs corresponding to given project items.

Parameters
names (*Iterable of str*) – item names to execute

execute_project()

Executes all dags in the project.

_validate_dags(*dags*)

Validates dags and logs error messages.

Parameters
dags (*Iterable*) – dags to validate

Returns
validated dag

Return type
list

stop()

Stops execution.

notify_resource_changes_to_predecessors(*item*)

Updates resources for direct predecessors of given item.

Parameters
item (ProjectItem) – item whose resources have changed

_update_incoming_connection_and_jump_resources(*item_name*, *trigger_resources*)

notify_resource_changes_to_successors(*item*)

Updates resources for direct successors and outgoing connections of given item.

Parameters
item (ProjectItem) – item whose resources have changed

_update_outgoing_connection_and_jump_resources(*item_name*, *trigger_resources*)

_notify_resource_changes(*trigger_name*, *target_names*, *provider_connections*, *update_resources*, *trigger_resources*)

Updates resources in given direction for immediate neighbours of an item.

Parameters

- **trigger_name** (*str*) – item whose resources have changed
- **target_names** (*Iterable of str*) – items to be notified
- **provider_connections** (*Callable*) – function that receives a target item name and returns a list of Connections from resource providers
- **update_resources** (*Callable*) – function that takes an item name, a list of provider names, and a dictionary of resources, and does the updating
- **trigger_resources** (*list of ProjectItemResource*) – resources from the trigger item

notify_resource_replacement_to_successors(*item*, *old*, *new*)

Replaces resources for direct successors and outgoing connections of given item.

Parameters

- **item** (ProjectItem) – item whose resources have changed
- **old** (*list of ProjectItemResource*) – old resource
- **new** (*list of ProjectItemResource*) – new resource

notify_resource_replacement_to_predecessors(*item*, *old*, *new*)

Replaces resources for direct predecessors.

Parameters

- **item** (ProjectItem) – item whose resources have changed
- **old** (*list of ProjectItemResource*) – old resources
- **new** (*list of ProjectItemResource*) – new resources

_update_item_resources(*target_item*, *direction*)

Updates up or downstream resources for a single project item. Called in both directions after removing a Connection.

Parameters

- **target_item** (ProjectItem) – item whose resource need update
- **direction** (*ExecutionDirection*) – FORWARD updates resources from upstream, BACKWARD from downstream

predecessor_names(*name*)

Collects direct predecessor item names.

Parameters
name (*str*) – name of the project item whose predecessors to collect

Returns
direct predecessor names

Return type
set of str

successor_names(*name*)

Collects direct successor item names.

Parameters
name (*str*) – name of the project item whose successors to collect

Returns
direct successor names

Return type
set of str

_outgoing_connections(*name*)

Collects outgoing connections.

Parameters
name (*str*) – name of the project item whose connections to collect

Returns
outgoing connections

Return type
set of Connection

_outgoing_jumps(*name*)

Collects outgoing jumps.

Parameters
name (*str*) – name of the project item whose jumps to collect

Returns
outgoing jumps

Return type
set of Jump

_outgoing_connections_and_jumps(*name*)

Collects outgoing connections and jumps.

Parameters
name (*str*) – name of the project item whose connections and jumps to collect

Returns
outgoing connections and jumps

Return type
set of Connection/Jump

incoming_connections(*name*)

Collects incoming connections.

Parameters
name (*str*) – name of the project item whose connections to collect

Returns
incoming connections

Return type
set of Connection

_incoming_jumps(*name*)

Collects incoming jumps.

Parameters
name (*str*) – name of the project item whose jumps to collect

Returns
incoming jumps

Return type
set of Jump

_incoming_connections_and_jumps(*name*)

Collects incoming connections and jumps.

Parameters
name (*str*) – name of the project item whose connections and jumps to collect

Returns
incoming connections

Return type
set of Connection/Jump

_update_successor(*successor*, *incoming_connections*, *resource_cache*)

_update_predecessor(*predecessor*, *outgoing_connections*, *resource_cache*)

_is_dag_valid(*dag*)

_update_ranks(*dag*)

prepare_remote_execution()

Pings the server and sends the project as a zip-file to server.

Returns
Job Id if server is ready for remote execution, empty string if something went wrong or "1" if local execution is enabled.

Return type
str

tear_down()

Cleans up project.

spinetoolbox.project.**node_successors**(*g*)

Returns a dict mapping nodes in topological order to a list of successors.

Parameters
g (*nx.DiGraph*) –

Returns
dict

spinetoolbox.project.**_edges_causing_loops**(*g*)

Returns a list of edges whose removal from g results in it becoming acyclic.

Parameters
g (*nx.DiGraph*) –

Returns
list

spinetoolbox.project.**_ranks**(*node_successors*)

Calculates node ranks.

Parameters
node_successors (*dict*) – a mapping from successor name to a list of predecessor names

Returns
a mapping from node name to rank

Return type
dict

spinetoolbox.project_commands

QUndoCommand subclasses for modifying the project.

authors

M. Marin (KTH)

date
12.2.2020

Module Contents

Classes

SpineToolboxCommand	
SetItemSpecificationCommand	Command to set the specification for a Tool.
MoveIconCommand	Command to move icons in the Design view.
SetProjectNameAndDescriptionCommand	Command to set the project name.
AddProjectItemsCommand	Command to add items.
RemoveAllProjectItemsCommand	Command to remove all items from project.
RemoveProjectItemsCommand	Command to remove items.
RenameProjectItemCommand	Command to rename project items.
AddConnectionCommand	Command to add connection between project items.
RemoveConnectionsCommand	Command to remove links.
AddJumpCommand	Command to add a jump between project items.
RemoveJumpsCommand	Command to remove jumps.
SetJumpConditionCommand	Command to set jump condition.
UpdateJumpCmdLineArgsCommand	Command to update Jump command line args.
SetFiltersOnlineCommand	Command to toggle filter value.
SetConnectionOptionsCommand	Command to set connection options.
AddSpecificationCommand	Command to add item specification to a project.
ReplaceSpecificationCommand	Command to replace item specification in project.
RemoveSpecificationCommand	Command to remove specs from a project.
SaveSpecificationAsCommand	Command to remove item specs from a project.

class spinetoolbox.project_commands.**SpineToolboxCommand**

Bases: PySide2.QtWidgets.QUndoCommand

property is_critical

Returns True if this command needs to be undone before closing the project without saving changes.

successfully_undone = False

Flag to register the outcome of undoing a critical command, so toolbox can react afterwards.

class spinetoolbox.project_commands.**SetItemSpecificationCommand**(*item*, *spec*, *old_spec*)

Bases: *SpineToolboxCommand*

Command to set the specification for a Tool.

Parameters

- **item** (ProjectItem) – the Item
- **spec** (*ProjectItemSpecification*) – the new spec
- **old_spec** (*ProjectItemSpecification*) – the old spec

redo()

undo()

class spinetoolbox.project_commands.**MoveIconCommand**(*icon*, *project*)

Bases: *SpineToolboxCommand*

Command to move icons in the Design view.

Parameters

- **icon** (ProjectItemIcon) – the icon
- **project** (SpineToolboxProject) – project

redo()

undo()

_move_to(*positions*)

class spinetoolbox.project_commands.**SetProjectNameAndDescriptionCommand**(*project*, *name*, *description*)

Bases: *SpineToolboxCommand*

Command to set the project name.

Parameters

- **project** (SpineToolboxProject) – the project
- **name** (*str*) – The new name
- **description** (*str*) – The new description

redo()

undo()

class spinetoolbox.project_commands.**AddProjectItemsCommand**(*project*, *items_dict*, *item_factories*, *silent=True*)

Bases: *SpineToolboxCommand*

Command to add items.

Parameters

- **project** (SpineToolboxProject) – the project
- **items_dict** (*dict*) – a mapping from item name to item dict
- **item_factories** (*dict*) – a mapping from item type to ProjectItemFactory
- **silent** (*bool*) – If True, suppress messages

redo()

undo()

class spinetoolbox.project_commands.**RemoveAllProjectItemsCommand**(*project*, *item_factories*, *delete_data=False*)

Bases: *SpineToolboxCommand*

Command to remove all items from project.

Parameters

- **project** (SpineToolboxProject) – the project
- **item_factories** (*dict*) – a mapping from item type to ProjectItemFactory
- **delete_data** (*bool*) – If True, deletes the directories and data associated with the items

redo()

undo()

class spinetoolbox.project_commands.**RemoveProjectItemsCommand**(*project*, *item_factories*, *item_names*, *delete_data=False*)

Bases: *SpineToolboxCommand*

Command to remove items.

Parameters

- **project** (SpineToolboxProject) – The project
- **item_factories** (*dict*) – a mapping from item type to ProjectItemFactory
- **item_names** (*list of str*) – Item names
- **delete_data** (*bool*) – If True, deletes the directories and data associated with the item

redo()

undo()

class spinetoolbox.project_commands.**RenameProjectItemCommand**(*project*, *previous_name*, *new_name*)

Bases: *SpineToolboxCommand*

Command to rename project items.

Parameters

- **project** (SpineToolboxProject) – the project
- **previous_name** (*str*) – item's previous name
- **new_name** (*str*) – the new name

property is_critical

Returns True if this command needs to be undone before closing the project without saving changes.

redo()

undo()

class spinetoolbox.project_commands.**AddConnectionCommand**(*project*, *source_name*, *source_position*, *destination_name*, *destination_position*)

Bases: *SpineToolboxCommand*

Command to add connection between project items.

Parameters

- **project** (SpineToolboxProject) – project
- **source_name** (*str*) – source item's name
- **source_position** (*str*) – link's position on source item's icon
- **destination_name** (*str*) – destination item's name
- **destination_position** (*str*) – link's position on destination item's icon

redo()

undo()

class spinetoolbox.project_commands.**RemoveConnectionsCommand**(*project*, *connections*)

Bases: *SpineToolboxCommand*

Command to remove links.

Parameters

- **project** (SpineToolboxProject) – project
- **connections** (*list of* LoggingConnection) – the connections

redo()

undo()

class spinetoolbox.project_commands.**AddJumpCommand**(*project*, *source_name*, *source_position*, *destination_name*, *destination_position*)

Bases: *SpineToolboxCommand*

Command to add a jump between project items.

Parameters

- **project** (SpineToolboxProject) – project
- **source_name** (*str*) – source item's name
- **source_position** (*str*) – link's position on source item's icon
- **destination_name** (*str*) – destination item's name
- **destination_position** (*str*) – link's position on destination item's icon

redo()

undo()

class spinetoolbox.project_commands.**RemoveJumpsCommand**(*project*, *jumps*)

Bases: *SpineToolboxCommand*

Command to remove jumps.

Parameters

- **project** (SpineToolboxProject) – project
- **jumps** (*list of* LoggingJump) – the jumps

redo()

undo()

class spinetoolbox.project_commands.**SetJumpConditionCommand**(*jump_properties*, *jump*, *condition*)

Bases: *SpineToolboxCommand*

Command to set jump condition.

Parameters

- **jump_properties** (JumpPropertiesWidget) – jump's properties tab
- **jump** (*Jump*) – target jump
- **condition** (*str*) – jump condition

redo()

undo()

class spinetoolbox.project_commands.**UpdateJumpCmdLineArgsCommand**(*jump_properties*, *jump*, *cmd_line_args*)

Bases: *SpineToolboxCommand*

Command to update Jump command line args.

Parameters

- **jump_properties** (JumpPropertiesWidget) – the item
- **cmd_line_args** (*list*) – list of command line args

redo()

undo()

class spinetoolbox.project_commands.**SetFiltersOnlineCommand**(*resource_filter_model*, *resource*, *filter_type*, *online*)

Bases: *SpineToolboxCommand*

Command to toggle filter value.

Parameters

- **resource_filter_model** (ResourceFilterModel) – filter model
- **resource** (*str*) – resource label
- **filter_type** (*str*) – filter type identifier
- **online** (*dict*) – mapping from scenario/tool id to online flag

redo()

undo()

class spinetoolbox.project_commands.**SetConnectionOptionsCommand**(*connection*, *options*)

Bases: *SpineToolboxCommand*

Command to set connection options.

Parameters

- **connection** (LoggingConnection) –
- **options** (*dict*) – containing options to be set

redo()

undo()

class spinetoolbox.project_commands.**AddSpecificationCommand**(*project*, *specification*, *save_to_disk*)

Bases: *SpineToolboxCommand*

Command to add item specification to a project.

Parameters

- **project** (ToolboxUI) – the toolbox
- **specification** (*ProjectItemSpecification*) – the spec
- **save_to_disk** (*bool*) – If True, save the specification to disk

redo()

undo()

class spinetoolbox.project_commands.**ReplaceSpecificationCommand**(*project*, *name*, *specification*)

Bases: *SpineToolboxCommand*

Command to replace item specification in project.

Parameters

- **project** (ToolboxUI) – the toolbox
- **name** (*str*) – the name of the spec to be replaced
- **specification** (*ProjectItemSpecification*) – the new spec

property is_critical

Returns True if this command needs to be undone before closing the project without saving changes.

redo()

undo()

class spinetoolbox.project_commands.**RemoveSpecificationCommand**(*project*, *name*)

Bases: *SpineToolboxCommand*

Command to remove specs from a project.

Parameters

- **project** (SpineToolboxProject) – the project
- **name** (*str*) – specification's name

redo()

undo()

class spinetoolbox.project_commands.**SaveSpecificationAsCommand**(*project*, *name*, *path*)

Bases: *SpineToolboxCommand*

Command to remove item specs from a project.

Parameters

- **project** (SpineToolboxProject) – the project
- **name** (*str*) – specification's name
- **path** (*str*) – new specification file location

redo()

undo()

spinetoolbox.project_item_icon

Classes for drawing graphics items on QGraphicsScene.

authors

M. Marin (KTH), P. Savolainen (VTT)

date

4.4.2018

Module Contents

Classes

ProjectItemIcon	Base class for project item icons drawn in Design View.
ConnectorButton	Connector button graphics item. Used for Link drawing between project items.
ExecutionIcon	An icon to show information about the item's execution.
ExclamationIcon	An icon to notify that a ProjectItem is missing some configuration.
RankIcon	An icon to show the rank of a ProjectItem within its DAG.

class spinetoolbox.project_item_icon.**ProjectItemIcon**(*toolbox*, *icon_file*, *icon_color*)

Bases: PySide2.QtWidgets.QGraphicsPathItem

Base class for project item icons drawn in Design View.

Parameters

- **toolbox** (ToolboxUI) – QMainWindow instance
- **icon_file** (*str*) – Path to icon resource
- **icon_color** (*QColor*) – Icon's color

ITEM_EXTENT = 64

FONT_SIZE_PIXELS = 12

rect()

_update_path()

update_path(*rounded*)

_do_update_path(*rounded*)

finalize(*name*, *x*, *y*)

Names the icon and moves it by given amount.

Parameters

- **name** (*str*) – icon's name
- **x** (*int*) – horizontal offset
- **y** (*int*) – vertical offset

_setup()

Setup item's attributes.

name()

Returns name of the item that is represented by this icon.

Returns
icon's name

Return type
str

update_name_item(*new_name*)

Set a new text to name item.

Parameters
new_name (*str*) – icon's name

set_name_attributes()

Set name QGraphicsSimpleTextItem attributes (font, size, position, etc.)

_reposition_name_item()

Set name item position (centered on top of the master icon).

conn_button(*position='left'*)

Returns item's connector button.

Parameters
position (*str*) – "left", "right" or "bottom"

Returns
connector button

Return type
QWidget

outgoing_connection_links()

Collects outgoing connection links.

Returns
outgoing links

Return type
list of *LinkBase*

incoming_links()

Collects incoming connection links.

Returns
outgoing links

Return type
list of *LinkBase*

_closest_connector(*pos*)

Returns the closest connector button to given scene pos.

_update_link_drawer_destination(*pos=None*)

Updates link drawer destination. If pos is None, then the link drawer would have no destination. Otherwise, the destination would be the connector button closest to pos.

hoverEnterEvent(*event*)

Sets a drop shadow effect to icon when mouse enters its boundaries.

Parameters
event (*QGraphicsSceneMouseEvent*) – Event

hoverMoveEvent(*event*)

hoverLeaveEvent(*event*)

Disables the drop shadow when mouse leaves icon boundaries.

Parameters
event (*QGraphicsSceneMouseEvent*) – Event

mousePressEvent(*event*)

Updates scene's icon group.

update_links_geometry()

Updates geometry of connected links to reflect this item's most recent position.

mouseReleaseEvent(*event*)

Clears pre-bump rects, and pushes a move icon command if necessary.

notify_item_move()

contextMenuEvent(*event*)

Show item context menu.

Parameters
event (*QGraphicsSceneMouseEvent*) – Mouse event

itemChange(*change*, *value*)

Reacts to item removal and position changes.

In particular, destroys the drop shadow effect when the items is removed from a scene and keeps track of item's movements on the scene.

Parameters

- **change** (*GraphicsItemChange*) – a flag signalling the type of the change
- **value** – a value related to the change

Returns
Whatever super() does with the value parameter

set_pos_without_bumping(*pos*)

Sets position without bumping other items. Needed for undoing move operations.

Parameters
pos (*QPointF*) –

_handle_collisions()

Handles collisions with other items.

make_room_for_item(*other*)

Makes room for another item.

Parameters
item (ProjectItemIcon) –

_restablish_bumped_items()

Moves bumped items back to their original position if no collision would happen anymore.

select_item()

Update GUI to show the details of the selected item.

paint(*painter*, *option*, *widget=None*)

Sets a dashed pen if selected.

class spinetoolbox.project_item_icon.**ConnectorButton**(*toolbox*, *parent*, *position='left'*)

Bases: PySide2.QtWidgets.QGraphicsPathItem

Connector button graphics item. Used for Link drawing between project items.

Parameters

- **toolbox** (ToolboxUI) – QMainWindow instance
- **parent** (ProjectItemIcon) – parent graphics item
- **position** (*str*) – Either "top", "left", "bottom", or "right"

property parent

brush

hover_brush

rect()

update_path(*parent_radius*)

outgoing_links()

incoming_links()

parent_name()

Returns project item name owning this connector button.

project_item()

Returns the project item this connector button is attached to.

Returns
project item

Return type
ProjectItem

mousePressEvent(*event*)

Connector button mouse press event.

Parameters
event (*QGraphicsSceneMouseEvent*) – Event

mouseReleaseEvent(*event*)

Connector button mouse release event.

Parameters
event (*QGraphicsSceneMouseEvent*) – Event

_start_link(*event*)

set_friend_connectors_enabled(*enabled*)

Enables or disables all connectors in the parent.

This is called by LinkDrawer to disable invalid connectors while drawing and reenabling them back when done.

Parameters
enabled (*bool*) – True to enable connectors, False to disable

set_hover_brush()

set_normal_brush()

hoverEnterEvent(*event*)

Sets a darker shade to connector button when mouse enters its boundaries.

Parameters
event (*QGraphicsSceneMouseEvent*) – Event

hoverLeaveEvent(*event*)

Restore original brush when mouse leaves connector button boundaries.

Parameters
event (*QGraphicsSceneMouseEvent*) – Event

itemChange(*change*, *value*)

If this is being removed from the scene while it's the origin of the link drawer, put the latter to sleep.

class spinetoolbox.project_item_icon.**ExecutionIcon**(*parent*)

Bases: PySide2.QtWidgets.QGraphicsEllipseItem

An icon to show information about the item's execution.

Parameters
parent (ProjectItemIcon) – the parent item

_CHECK =

_CROSS =

_CLOCK =

_SKIP =

item_name()

_repaint(*text*, *color*)

mark_execution_waiting()

mark_execution_started()

mark_execution_finished(*item_finish_state*)

hoverEnterEvent(*event*)

hoverLeaveEvent(*event*)

class spinetoolbox.project_item_icon.**ExclamationIcon**(*parent*)

Bases: PySide2.QtWidgets.QGraphicsTextItem

An icon to notify that a ProjectItem is missing some configuration.

Parameters
parent (ProjectItemIcon) – the parent item

FONT_SIZE_PIXELS = 14

clear_notifications()

Clear all notifications.

add_notification(*text*)

Add a notification.

remove_notification(*subtext*)

Remove the first notification that includes given subtext.

hoverEnterEvent(*event*)

Shows notifications as tool tip.

Parameters
event (*QGraphicsSceneMouseEvent*) – Event

hoverLeaveEvent(*event*)

Hides tool tip.

Parameters
event (*QGraphicsSceneMouseEvent*) – Event

class spinetoolbox.project_item_icon.**RankIcon**(*parent*)

Bases: PySide2.QtWidgets.QGraphicsTextItem

An icon to show the rank of a ProjectItem within its DAG.

Parameters
parent (ProjectItemIcon) – the parent item

_make_path(*radius*)

update_path(*radius*)

set_rank(*rank*)

spinetoolbox.project_upgrader

Contains ProjectUpgrader class used in upgrading and converting projects and project dicts from earlier versions to the latest version.

authors

P. Savolainen (VTT)

date
8.11.2019

Module Contents

Classes

ProjectUpgrader	Class to upgrade/convert projects from earlier versions to the current version.

Functions

_fix_1d_array_to_array(mappings)	Replaces '1d array' with 'array' for parameter type in Importer mappings.

class spinetoolbox.project_upgrader.**ProjectUpgrader**(*toolbox*)

Class to upgrade/convert projects from earlier versions to the current version.

Parameters
toolbox (ToolboxUI) – App main window instance

upgrade(*project_dict*, *project_dir*)

Upgrades the project described in given project dictionary to the latest version.

Parameters

- **project_dict** (*dict*) – Project configuration dictionary
- **project_dir** (*str*) – Path to current project directory

Returns
Latest version of the project info dictionary

Return type
dict

upgrade_to_latest(*v*, *project_dict*, *project_dir*)

Upgrades the given project dictionary to the latest version.

Parameters

- **v** (*int*) – Current version of the project dictionary
- **project_dict** (*dict*) – Project dictionary (JSON) to be upgraded
- **project_dir** (*str*) – Path to current project directory

Returns
Upgraded project dictionary

Return type
dict

static upgrade_v1_to_v2(*old*, *factories*)

Upgrades version 1 project dictionary to version 2.

Changes:
objects -> items, tool_specifications -> specifications store project item dicts under [“items”][<project item name>] instead of using their categories as keys specifications must be a dict instead of a list Add specifications[“Tool”] that must be a dict Remove “short name” from all project items

Parameters

- **old** (*dict*) – Version 1 project dictionary
- **factories** (*dict*) – Mapping of item type to item factory

Returns
Version 2 project dictionary

Return type
dict

upgrade_v2_to_v3(*old*, *project_dir*, *factories*)

Upgrades version 2 project dictionary to version 3.

Changes:

1. Move "specifications" from "project" -> "Tool" to just "project"
2. The "mappings" from importer items are used to build Importer specifications

Parameters

- **old** (*dict*) – Version 2 project dictionary
- **project_dir** (*str*) – Path to current project directory
- **factories** (*dict*) – Mapping of item type to item factory

Returns
Version 3 project dictionary

Return type
dict

static upgrade_v3_to_v4(*old*)

Upgrades version 3 project dictionary to version 4.

Changes:

1. Rename "Exporter" item type to "GdxExporter"

Parameters
old (*dict*) – Version 3 project dictionary

Returns
Version 4 project dictionary

Return type
dict

static upgrade_v4_to_v5(*old*)

Upgrades version 4 project dictionary to version 5.

Changes:

1. Get rid of "Combiner" items.

Parameters
old (*dict*) – Version 4 project dictionary

Returns
Version 5 project dictionary

Return type
dict

`static upgrade_v5_to_v6(old, project_dir)`

Upgrades version 5 project dictionary to version 6.

Changes:

1. Data store URL labels do not have '{' and '}' anymore
2. Importer stores resource labels instead of serialized paths in "file_selection".
3. Gimlet's "selections" is now called "file_selection"
4. Gimlet stores resource labels instead of serialized paths in "file_selection".
5. Gimlet and Tool store command line arguments as serialized CmdLineArg objects, not serialized paths

Parameters

- **old** (*dict*) – Version 5 project dictionary
- **project_dir** (*str*) – Path to current project directory

Returns
Version 6 project dictionary

Return type
dict

`static upgrade_v6_to_v7(old)`

Upgrades version 6 project dictionary to version 7.

Changes:

1. Introduces Mergers in between DS -> DS links.

Parameters
old (*dict*) – Version 6 project dictionary

Returns
Version 7 project dictionary

Return type
dict

`static upgrade_v7_to_v8(old)`

Upgrades version 7 project dictionary to version 8.

Changes:

1. Move purge settings from items to their outgoing connections.

Parameters
old (*dict*) – Version 7 project dictionary

Returns
Version 8 project dictionary

Return type
dict

static make_unique_importer_specification_name(*importer_name*, *label*, *k*)

get_project_directory()

Asks the user to select a new project directory. If the selected directory is already a Spine Toolbox project directory, asks if overwrite is ok. Used when opening a project from an old style project file (.proj).

Returns
Path to project directory or an empty string if operation is canceled.

Return type
str

is_valid(*v*, *p*)

Checks given project dict if it is valid for given version.

is_valid_v1(*p*)

Checks that the given project JSON dictionary contains a valid version 1 Spine Toolbox project. Valid meaning, that it contains all required keys and values are of the correct type.

Parameters
p (*dict*) – Project information JSON

Returns
True if project is a valid version 1 project, False if it is not

Return type
bool

is_valid_v2_to_8(*p*, *v*)

Checks that the given project JSON dictionary contains a valid version 2 to 6 Spine Toolbox project. Valid meaning, that it contains all required keys and values are of the correct type.

Parameters

- **p** (*dict*) – Project information JSON
- **v** (*int*) – Version

Returns
True if project is a valid version 2 project, False if it is not

Return type
bool

backup_project_file(*project_dir*, *v*)

Makes a backup copy of project.json file.

force_save(*p*, *project_dir*)

Saves given project dictionary to project.json file. Used to force save project.json file when the project dictionary has been upgraded.

spinetoolbox.project_upgrader.**_fix_1d_array_to_array**(*mappings*)

Replaces '1d array' with 'array' for parameter type in Importer mappings.

With spinedb_api >= 0.3, '1d array' parameter type was replaced by 'array'. Other settings in a mapping are backwards compatible except the name.

spinetoolbox.qthread_pool_executor

Qt-based thread pool executor.

authors

M. Marin (ER)

date

25.10.2022

Module Contents

Classes

QtBasedQueue	A Qt-based clone of queue.Queue.
QtBasedFuture	A Qt-based clone of concurrent.futures.Future.
QtBasedThread	A Qt-based clone of threading.Thread.
QtBasedThreadPoolExecutor	A Qt-based clone of concurrent.futures.ThreadPoolExecutor

exception spinetoolbox.qthread_pool_executor.**TimeOutError**

Bases: `Exception`

An exception to raise when a timeouts expire

Initialize self. See help(type(self)) for accurate signature.

class spinetoolbox.qthread_pool_executor.**QtBasedQueue**

A Qt-based clone of queue.Queue.

put(*item*)

get(*timeout=None*)

class spinetoolbox.qthread_pool_executor.**QtBasedFuture**

A Qt-based clone of concurrent.futures.Future.

set_result(*result*)

set_exception(*exc*)

result(*timeout=None*)

exception(*timeout=None*)

class spinetoolbox.qthread_pool_executor.**QtBasedThread**(*target=None*, *args=()*)

Bases: `PySide2.QtCore.QThread`

A Qt-based clone of threading.Thread.

run()

class spinetoolbox.qthread_pool_executor.**QtBasedThreadPoolExecutor**(*max_workers=None*)

A Qt-based clone of concurrent.futures.ThreadPoolExecutor

submit(*fn*, **args*, ***kwargs*)

_spawn_thread()

_do_work()

shutdown()

spinetoolbox.spine_db_commands

QUndoCommand subclasses for modifying the db.

authors

M. Marin (KTH)

date

31.1.2020

Module Contents

Classes

AgedUndoStack	
AgedUndoCommand	**param parent** The parent command, used for defining macros.
SpineDBCommand	**param db_mngr** SpineDBManager instance
AddItemsCommand	**param db_mngr** SpineDBManager instance
UpdateItemsCommand	**param db_mngr** SpineDBManager instance
RemoveItemsCommand	**param db_mngr** SpineDBManager instance

class spinetoolbox.spine_db_commands.**AgedUndoStack**

Bases: PySide2.QtWidgets.QUndoStack

property redo_age

property undo_age

commands()

class spinetoolbox.spine_db_commands.**AgedUndoCommand**(*parent=None*)

Bases: PySide2.QtWidgets.QUndoCommand

Parameters

parent (*QUndoCommand, optional*) – The parent command, used for defining macros.

property age

redo()

undo()

class spinetoolbox.spine_db_commands.**SpineDBCommand**(*db_mngr*, *db_map*, *parent=None*)

Bases: *AgedUndoCommand*

Parameters

- **db_mngr** (SpineDBManager) – SpineDBManager instance
- **db_map** (*DiffDatabaseMapping*) – DiffDatabaseMapping instance
- **parent** (*QUndoCommand, optional*) – The parent command, used for defining macros.

_add_command_name

_update_command_name

_add_method_name

_update_method_name

_added_signal_name

_updated_signal_name

static redomethod(*func*)

Returns a new redo method that determines if the command was completed. The command is completed if calling the function triggers the `completed_signal`. Once the command is completed, we don't listen to the signal anymore and we also silence the affected Spine db editors. If the signal is not received, then the command is declared obsolete.

static undomethod(*func*)

Returns a new undo method that silences the affected Spine db editors.

abstract receive_items_changed(_)

class spinetoolbox.spine_db_commands.**AddItemsCommand**(*db_mngr*, *db_map*, *data*, *item_type*, *parent=None*, *check=True*)

Bases: *SpineDBCommand*

Parameters

- **db_mngr** (SpineDBManager) – SpineDBManager instance
- **db_map** (*DiffDatabaseMapping*) – DiffDatabaseMapping instance
- **data** (*list*) – list of dict-items to add
- **item_type** (*str*) – the item type
- **parent** (*QUndoCommand, optional*) – The parent command, used for defining macros.

redo()

undo()

receive_items_changed(*db_map_data*)

class spinetoolbox.spine_db_commands.**UpdateItemsCommand**(*db_mngr*, *db_map*, *data*, *item_type*, *parent=None*, *check=True*)

Bases: *SpineDBCommand*

Parameters

- **db_mngr** (SpineDBManager) – SpineDBManager instance
- **db_map** (*DiffDatabaseMapping*) – DiffDatabaseMapping instance
- **data** (*list*) – list of dict-items to update
- **item_type** (*str*) – the item type
- **parent** (*QUndoCommand, optional*) – The parent command, used for defining macros.

_undo_item(*db_map*, *id_*)

redo()

undo()

receive_items_changed(*db_map_data*)

class spinetoolbox.spine_db_commands.**RemoveItemsCommand**(*db_mngr*, *db_map*, *typed_data*, *parent=None*)

Bases: *SpineDBCommand*

Parameters

- **db_mngr** (SpineDBManager) – SpineDBManager instance
- **db_map** (*DiffDatabaseMapping*) – DiffDatabaseMapping instance
- **typed_data** (*dict*) – lists of dict-items to remove keyed by string type
- **parent** (*QUndoCommand, optional*) – The parent command, used for defining macros.

redo()

undo()

receive_items_changed(*typed_db_map_data*)

spinetoolbox.spine_db_icon_manager

Provides SpineDBIconManager.

authors

M. Marin (KTH)

date

3.2.2021

Module Contents

Classes

_SceneSvgRenderer	
SpineDBIconManager	A class to manage object_class icons for spine db editors.
SceneIconEngine	Specialization of QIconEngine used to draw scene-based icons.

Functions

_align_text_in_item(item)	
_center_scene(scene)	

spinetoolbox.spine_db_icon_manager.**_align_text_in_item**(*item*)

spinetoolbox.spine_db_icon_manager.**_center_scene**(*scene*)

class spinetoolbox.spine_db_icon_manager.**_SceneSvgRenderer**

Bases: PySide2.QtSvg.QSvgRenderer

scene

classmethod from_scene(*scene*)

class spinetoolbox.spine_db_icon_manager.**SpineDBIconManager**

A class to manage object_class icons for spine db editors.

update_icon_caches(*classes*)

Called after adding or updating entity classes. Stores display_icons and clears obsolete entries from the relationship class and entity group renderer caches.

_create_icon_renderer(*icon_code*, *color_code*)

icon_renderer(*icon_code*, *color_code*)

_create_class_renderer(*class_name*)

class_renderer(*class_name*)

_create_rel_cls_renderer(*object_class_names*)

relationship_class_renderer(*rel_cls_name*, *str_object_class_name_list*)

_create_group_renderer(*class_name*)

group_renderer(*class_name*)

static icon_from_renderer(*renderer*)

class spinetoolbox.spine_db_icon_manager.**SceneIconEngine**(*scene*)

Bases: *spinetoolbox.helpers.TransparentIconEngine*

Specialization of QIconEngine used to draw scene-based icons.

paint(*painter*, *rect*, *mode=None*, *state=None*)

spinetoolbox.spine_db_manager

The SpineDBManager class

authors

P. Vennström (VTT) and M. Marin (KTH)

date

2.10.2019

Module Contents

Classes

SpineDBManager	Class to manage DBs within a project.

Functions

do_create_new_spine_database(url)	Creates a new spine database at the given url.

spinetoolbox.spine_db_manager.**do_create_new_spine_database**(*url*)

Creates a new spine database at the given url.

class spinetoolbox.spine_db_manager.**SpineDBManager**(*settings*, *parent*)

Bases: PySide2.QtCore.QObject

Class to manage DBs within a project.

Initializes the instance.

Parameters

- **settings** (*QSettings*) – Toolbox settings
- **parent** (*QObject, optional*) – parent object

property db_maps

property db_urls

error_msg

session_refreshed

session_committed

session_rolled_back

scenarios_added

alternatives_added

object_classes_added

objects_added

relationship_classes_added

relationships_added

entity_groups_added

parameter_definitions_added

parameter_values_added

parameter_value_lists_added

list_values_added

features_added

tools_added

tool_features_added

tool_feature_methods_added

metadata_added

entity_metadata_added

parameter_value_metadata_added

scenarios_removed

alternatives_removed

object_classes_removed

objects_removed

relationship_classes_removed

relationships_removed

entity_groups_removed

parameter_definitions_removed

parameter_values_removed

parameter_value_lists_removed

list_values_removed

features_removed

tools_removed

tool_features_removed

tool_feature_methods_removed

metadata_removed

entity_metadata_removed

parameter_value_metadata_removed

items_removed

scenarios_updated

alternatives_updated

object_classes_updated

objects_updated

relationship_classes_updated

relationships_updated

parameter_definitions_updated

parameter_values_updated

parameter_value_lists_updated

list_values_updated

features_updated

tools_updated

tool_features_updated

tool_feature_methods_updated

metadata_updated

entity_metadata_updated

parameter_value_metadata_updated

items_removed_from_cache

scenario_alternatives_added

scenario_alternatives_updated

scenario_alternatives_removed

waiting_for_fetcher

fetcher_waiting_over

connect_signals()

Connects signals.

_get_worker(*db_map*)

Returns a worker.

Parameters
db_map (*DiffDatabaseMapping*) –

Returns
SpineDBWorker

can_fetch_more(*db_map*, *parent*)

Whether or not we can fetch more items of given type from given db.

Parameters

- **db_map** (*DiffDatabaseMapping*) –
- **parent** (FetchParent) – The object that requests the fetching.

Returns
bool

fetch_more(*db_map*, *parent*)

Fetches more items of given type from given db.

Parameters

- **db_map** (*DiffDatabaseMapping*) –
- **parent** (FetchParent) – The object that requests the fetching.

cache_items(*item_type*, *db_map_data*)

Caches data for a given type. It works for both insert and update operations.

Parameters

- **item_type** (*str*) –
- **db_map_data** (*dict*) – lists of dictionary items keyed by DiffDatabaseMapping

_pop_item(*db_map*, *item_type*, *id_*)

uncache_removed_items(*db_map_typed_ids*)

Removes data that has been removed from the database also from cache.

Parameters
db_map_typed_ids (*dict*) – items to remove

get_db_map_cache(*db_map*, *item_types=None*, *only_descendants=False*, *include_ancestors=False*)

get_icon_mngr(*db_map*)

Returns an icon manager for given db_map.

Parameters
db_map (*DiffDatabaseMapping*) –

Returns
SpineDBIconManager

update_icons(*db_map_data*)

Runs when object classes are added or updated. Setups icons for those classes.

Parameters
db_map_data (*dict*) – lists of dictionary items keyed by DiffDatabaseMapping

db_map(*url*)

Returns a database mapping for given URL.

Parameters
url (*str*) – a database URL

Returns
a database map or None if not found

Return type
DiffDatabaseMapping

create_new_spine_database(*url*, *logger*)

close_session(*url*)

Pops any db map on the given url and closes its connection.

Parameters
url (*str*) –

close_all_sessions()

Closes connections to all database mappings.

get_db_map(*url*, *logger*, *codename=None*, *upgrade=False*, *create=False*)

Returns a DiffDatabaseMapping instance from url if possible, None otherwise. If needed, asks the user to upgrade to the latest db version.

Parameters

- **url** (*str, URL*) –
- **logger** (LoggerInterface) –
- **codename** (*str, NoneType, optional*) –
- **upgrade** (*bool, optional*) –
- **create** (*bool, optional*) –

Returns
DiffDatabaseMapping, NoneType

_do_get_db_map(*url*, *codename*, *upgrade*, *create*)

Returns a memorized DiffDatabaseMapping instance from url. Called by *get_db_map*.

Parameters

- **url** (*str, URL*) –
- **codename** (*str, NoneType*) –
- **upgrade** (*bool*) –
- **create** (*bool*) –

Returns
DiffDatabaseMapping

query(*db_map*, *sq_name*)

For tests.

register_listener(*listener*, **db_maps*)

Register given listener for all given db_map's signals.

Parameters

- **listener** (*object*) –
- **db_maps** (*DiffDatabaseMapping*) –

unregister_listener(*listener*, **db_maps*, *commit_dirty=False*, *commit_msg=''*)

Unregisters given listener from given db_map signals. If any of the db_maps becomes an orphan and is dirty, prompts user to commit or rollback.

Parameters

- **listener** (*object*) –
- ***db_maps** (*DiffDatabaseMapping*) –
- **commit_dirty** (*bool*) – True to commit dirty database mapping, False to roll back
- **commit_msg** (*str*) – commit message

is_dirty(*db_map*)

Returns True if mapping has pending changes.

Parameters
db_map (*DiffDatabaseMapping*) – database mapping

Returns
True if db_map has pending changes, False otherwise

Return type
bool

dirty(**db_maps*)

Filters clean mappings from given database maps.

Parameters
***db_maps** – mappings to check

Returns
dirty mappings

Return type
list of DiffDatabaseMapping

dirty_and_without_editors(*listener*, **db_maps*)

Checks which of the given database mappings are dirty and have no editors.

Parameters

- **listener** (*Any*) – a listener object
- ***db_maps** – mappings to check

Returns
mappings that are dirty and don't have editors

Return type
list of DiffDatabaseMapping

clean_up()

refresh_session(**db_maps*)

commit_session(*commit_msg*, **dirty_db_maps*, *cookie=None*)

Commits the current session.

Parameters

- **commit_msg** (*str*) – commit message for all database maps
- ***dirty_db_maps** – dirty database maps to commit
- **cookie** (*object, optional*) – a free form identifier which will be forwarded to `session_committed` signal

notify_session_committed(*cookie*, **db_maps*)

Notifies manager and listeners when a commit has taken place by a third party.

Parameters

- **cookie** (*Any*) – commit cookie
- ***db_maps** – database maps that were committed

rollback_session(**dirty_db_maps*)

Rolls back the current session.

Parameters
***dirty_db_maps** – dirty database maps to commit

_restart_fetching(*db_maps*)

Restarts fetching

entity_class_renderer(*db_map*, *entity_type*, *entity_class_id*, *for_group=False*)

Returns an icon renderer for a given entity class.

Parameters

- **db_map** (*DiffDatabaseMapping*) – database map
- **entity_type** (*str*) – either 'object_class' or 'relationship_class'
- **entity_class_id** (*int*) – entity class' id
- **for_group** (*bool*) – if True, return the group object icon instead

Returns
requested renderer or None if no entity class was found

Return type
QSvgRenderer

entity_class_icon(*db_map*, *entity_type*, *entity_class_id*, *for_group=False*)

Returns an appropriate icon for a given entity class.

Parameters

- **db_map** (*DiffDatabaseMapping*) – database map
- **entity_type** (*str*) – either 'object_class' or 'relationship_class'
- **entity_class_id** (*int*) – entity class' id
- **for_group** (*bool*) – if True, return the group object icon instead

Returns
requested icon or None if no entity class was found

Return type
QIcon

get_item(*db_map*, *item_type*, *id_*, *only_visible=True*)

Returns the item of the given type in the given db map that has the given id, or an empty dict if not found.

Parameters

- **db_map** (*DiffDatabaseMapping*) –
- **item_type** (*str*) –
- **id** (*int*) –
- **only_visible** (*bool, optional*) – If True, only looks in items that have already made it into the cache.

Returns
cached item

Return type
CacheItem

get_field(*db_map*, *item_type*, *id_*, *field*, *only_visible=True*)

get_items(*db_map*, *item_type*, *only_visible=True*)

Returns a list of the items of the given type in the given db map.

Parameters

- **db_map** (*DiffDatabaseMapping*) –
- **item_type** (*str*) –
- **only_visible** (*bool, optional*) – If True, only returns items that have already made it into the cache.

Returns
list

get_items_by_field(*db_map*, *item_type*, *field*, *value*, *only_visible=True*)

Returns a list of items of the given type in the given db map that have the given value for the given field.

Parameters

- **db_map** (*DiffDatabaseMapping*) –
- **item_type** (*str*) –
- **field** (*str*) –
- **value** –

Returns
list

get_item_by_field(*db_map*, *item_type*, *field*, *value*, *only_visible=True*)

Returns the first item of the given type in the given db map that has the given value for the given field Returns an empty dictionary if none found.

Parameters

- **db_map** (*DiffDatabaseMapping*) –

- **item_type** (*str*) –
- **field** (*str*) –
- **value** –

Returns
dict

static display_data_from_parsed(*parsed_data*)

Returns the value's database representation formatted for Qt.DisplayRole.

static tool_tip_data_from_parsed(*parsed_data*)

Returns the value's database representation formatted for Qt.ToolTipRole.

get_value(*db_map*, *item_type*, *id_*, *role=Qt.DisplayRole*)

Returns the value or default value of a parameter.

Parameters

- **db_map** (*DiffDatabaseMapping*) –
- **item_type** (*str*) – either "parameter_definition", "parameter_value", or "list_value"
- **id** (*int*) – The parameter_value or definition id
- **role** (*int, optional*) –

Returns
any

get_value_from_data(*data*, *role=Qt.DisplayRole*)

Returns the value or default value of a parameter directly from data. Used by `EmptyParameterModel.data()`.

Parameters

- **data** (*str*) – joined value and type
- **role** (*int, optional*) –

Returns
any

static _parse_value(*db_value*, *value_type=None*)

_format_value(*parsed_value*, *role=Qt.DisplayRole*)

Formats the given value for the given role.

Parameters

- **parsed_value** (*object*) – A python object as returned by spinedb_api.from_database
- **role** (*int, optional*) –

get_value_indexes(*db_map*, *item_type*, *id_*)

Returns the value or default value indexes of a parameter.

Parameters

- **db_map** (*DiffDatabaseMapping*) –
- **item_type** (*str*) – either "parameter_definition" or "parameter_value"
- **id** (*int*) – The parameter_value or definition id

get_value_index(*db_map*, *item_type*, *id_*, *index*, *role=Qt.DisplayRole*)

Returns the value or default value of a parameter for a given index.

Parameters

- **db_map** (*DiffDatabaseMapping*) –
- **item_type** (*str*) – either "parameter_definition" or "parameter_value"
- **id** (*int*) – The parameter_value or definition id
- **index** – The index to retrieve
- **role** (*int, optional*) –

get_value_list_item(*db_map*, *id_*, *index*, *role=Qt.DisplayRole*, *only_visible=True*)

Returns one value item of a parameter_value_list.

Parameters

- **db_map** (*DiffDatabaseMapping*) –
- **id** (*int*) – The parameter_value_list id
- **index** (*int*) – The value item index
- **role** (*int, optional*) –

get_parameter_value_list(*db_map*, *id_*, *role=Qt.DisplayRole*, *only_visible=True*)

Returns a parameter_value_list formatted for the given role.

Parameters

- **db_map** (*DiffDatabaseMapping*) –
- **id** (*int*) – The parameter_value_list id
- **role** (*int, optional*) –

get_scenario_alternative_id_list(*db_map*, *scen_id*, *only_visible=True*)

import_data(*db_map_data*, *command_text='Import data'*)

Imports the given data into given db maps using the dedicated import functions from spinedb_api. Condenses all in a single command for undo/redo.

Parameters

- **db_map_data** (*dict(DiffDatabaseMapping, dict())*) – Maps dbs to data to be passed as keyword arguments to *get_data_for_import*
- **command_text** (*str, optional*) – What to call the command that condenses the operation.

add_or_update_items(*db_map_data*, *method_name*, *item_type*, *signal_name*, *readd=False*, *check=True*)

_add_or_update_items(*db_map_data*, *method_name*, *item_type*, *signal_name*, *check*)

_readd_items(*db_map_data*, *method_name*, *item_type*, *signal_name*)

add_alternatives(*db_map_data*)

Adds alternatives to db.

Parameters

db_map_data (*dict*) – lists of items to add keyed by DiffDatabaseMapping

add_scenarios(*db_map_data*)

Adds scenarios to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DiffDatabaseMapping

add_object_classes(*db_map_data*)

Adds object classes to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DiffDatabaseMapping

add_objects(*db_map_data*)

Adds objects to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DiffDatabaseMapping

add_object_metadata(*db_map_data*)

Adds object metadata to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DiffDatabaseMapping

add_relationship_classes(*db_map_data*)

Adds relationship classes to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DiffDatabaseMapping

add_relationships(*db_map_data*)

Adds relationships to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DiffDatabaseMapping

add_object_groups(*db_map_data*)

Adds object groups to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DiffDatabaseMapping

add_entity_groups(*db_map_data*)

Adds entity groups to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DiffDatabaseMapping

add_parameter_definitions(*db_map_data*)

Adds parameter definitions to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DiffDatabaseMapping

add_parameter_values(*db_map_data*)

Adds parameter values to db without checking integrity.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DiffDatabaseMapping

add_parameter_value_lists(*db_map_data*)

Adds parameter_value lists to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DiffDatabaseMapping

add_list_values(*db_map_data*)

Adds parameter_value list values to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DiffDatabaseMapping

add_features(*db_map_data*)

Adds features to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DiffDatabaseMapping

add_tools(*db_map_data*)

Adds tools to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DiffDatabaseMapping

add_tool_features(*db_map_data*)

Adds tool features to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DiffDatabaseMapping

add_tool_feature_methods(*db_map_data*)

Adds tool feature methods to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DiffDatabaseMapping

add_metadata(*db_map_data*)

Adds metadata to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DiffDatabaseMapping

add_entity_metadata(*db_map_data*)

Adds entity metadata to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DiffDatabaseMapping

add_parameter_value_metadata(*db_map_data*)

Adds parameter value metadata to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DiffDatabaseMapping

update_alternatives(*db_map_data*)

Updates alternatives in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DiffDatabaseMapping

update_scenarios(*db_map_data*)

Updates scenarios in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DiffDatabaseMapping

update_object_classes(*db_map_data*)

Updates object classes in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DiffDatabaseMapping

update_objects(*db_map_data*)

Updates objects in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DiffDatabaseMapping

update_relationship_classes(*db_map_data*)

Updates relationship classes in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DiffDatabaseMapping

update_relationships(*db_map_data*)

Updates relationships in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DiffDatabaseMapping

update_parameter_definitions(*db_map_data*)

Updates parameter definitions in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DiffDatabaseMapping

update_parameter_values(*db_map_data*)

Updates parameter values in db without checking integrity.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DiffDatabaseMapping

update_expanded_parameter_values(*db_map_data*)

Updates expanded parameter values in db without checking integrity.

Parameters
db_map_data (*dict*) – lists of expanded items to update keyed by DiffDatabaseMapping

update_parameter_value_lists(*db_map_data*)

Updates parameter_value lists in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DiffDatabaseMapping

update_list_values(*db_map_data*)

Updates parameter_value list values in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DiffDatabaseMapping

update_features(*db_map_data*)

Updates features in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DiffDatabaseMapping

update_tools(*db_map_data*)

Updates tools in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DiffDatabaseMapping

update_tool_features(*db_map_data*)

Updates tools features in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DiffDatabaseMapping

update_tool_feature_methods(*db_map_data*)

Updates tools feature methods in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DiffDatabaseMapping

update_metadata(*db_map_data*)

Updates metadata in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DiffDatabaseMapping

update_entity_metadata(*db_map_data*)

Updates entity metadata in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DiffDatabaseMapping

update_parameter_value_metadata(*db_map_data*)

Updates parameter value metadata in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DiffDatabaseMapping

set_scenario_alternatives(*db_map_data*)

Sets scenario alternatives in db.

Parameters
db_map_data (*dict*) – lists of items to set keyed by DiffDatabaseMapping

purge_items(*db_map_purgable_items*)

Purges selected items from given database.

Parameters
db_map_purgable_items (*dict*) – mapping from database map to list of purgable item types

remove_items(*db_map_typed_ids*)

Pushes a command to remove items to undo stack.

do_remove_items(*db_map_typed_ids*)

Removes items from database.

Parameters
db_map_typed_ids (*dict*) – mapping DiffDatabaseMapping to item type (str) to lists of items to remove

static db_map_ids(*db_map_data*)

static db_map_class_ids(*db_map_data*)

find_cascading_relationship_classes(*db_map_ids*, *only_visible=True*)
Finds and returns cascading relationship classes for the given object_class ids.

find_cascading_entities(*db_map_ids*, *item_type*, *only_visible=True*)
Finds and returns cascading entities for the given entity_class ids.

find_cascading_relationships(*db_map_ids*, *only_visible=True*)
Finds and returns cascading relationships for the given object ids.

find_cascading_parameter_data(*db_map_ids*, *item_type*, *only_visible=True*)
Finds and returns cascading parameter definitions or values for the given entity_class ids.

find_cascading_parameter_definitions_by_value_list(*db_map_ids*, *only_visible=True*)
Finds and returns cascading parameter definitions for the given parameter_value_list ids.

find_cascading_parameter_definitions_by_removed_value_list(*db_map_ids*, *only_visible=True*)
Finds and returns cascading parameter definitions for the given parameter_value_list ids that have been removed.

find_cascading_parameter_values_by_entity(*db_map_ids*, *only_visible=True*)
Finds and returns cascading parameter values for the given entity ids.

find_cascading_parameter_values_by_definition(*db_map_ids*, *only_visible=True*)
Finds and returns cascading parameter values for the given parameter_definition ids.

find_groups_by_entity(*db_map_ids*, *only_visible=True*)
Finds and returns groups for the given entity ids.

find_groups_by_member(*db_map_ids*, *only_visible=True*)
Finds and returns groups for the given entity ids.

find_cascading_parameter_values_by_alternative(*db_map_ids*, *only_visible=True*)
Finds and returns cascading parameter values for the given alternative ids.

find_cascading_features_by_parameter_definition(*db_map_ids*, *only_visible=True*)
Finds and returns cascading features for the given parameter definition ids.

find_cascading_features_by_parameter_value_list(*db_map_ids*, *only_visible=True*)
Finds and returns cascading features for the given parameter value list ids.

find_cascading_tool_features_by_feature(*db_map_ids*, *only_visible=True*)
Finds and returns cascading tool features for the given feature ids.

find_cascading_parameter_values_by_list_value(*db_map_ids*, *only_visible=True*)
Finds and returns cascading parameter values for the given list value ids.

find_cascading_parameter_definitions_by_list_value(*db_map_ids*, *only_visible=True*)
Finds and returns cascading parameter definitions for the given list value ids.

find_cascading_scenario_alternatives_by_scenario(*db_map_ids*, *only_visible=True*)
Finds and returns cascading scenario alternatives for the given scenario ids.

_refresh_parameter_value_lists(*db_map_data*)

Refreshes cached parameter value lists when updating list values.

Parameters
db_map_data (*dict*) – lists of updated items keyed by DiffDatabaseMapping

_refresh_scenario_alternatives(*db_map_data*)

Refreshes cached scenarios when updating scenario alternatives.

Parameters
db_map_data (*dict*) – lists of updated items keyed by DiffDatabaseMapping

_cascade_refresh_objects_by_group(*db_map_data*)

_cascade_refresh_relationships_by_group(*db_map_data*)

_cascade_refresh_entities_by_group(*db_map_data*, *item_type*, *updated_signal*)

_cascade_refresh_relationship_classes(*db_map_data*)

Refreshes cached relationship classes when updating object classes.

Parameters
db_map_data (*dict*) – lists of updated items keyed by DiffDatabaseMapping

_cascade_refresh_relationships_by_object(*db_map_data*)

Refreshed cached relationships in cascade when updating objects.

Parameters
db_map_data (*dict*) – lists of updated items keyed by DiffDatabaseMapping

_cascade_refresh_parameter_definitions(*db_map_data*)

Refreshes cached parameter definitions in cascade when updating entity classes.

Parameters
db_map_data (*dict*) – lists of updated items keyed by DiffDatabaseMapping

_cascade_refresh_parameter_definitions_by_value_list(*db_map_data*)

Refreshes cached parameter definitions when updating parameter_value lists.

Parameters
db_map_data (*dict*) – lists of updated items keyed by DiffDatabaseMapping

_cascade_refresh_parameter_definitions_by_removed_value_list(*db_map_data*)

Refreshes cached parameter definitions when removing parameter_value lists.

Parameters
db_map_data (*dict*) – lists of removed items keyed by DiffDatabaseMapping

_cascade_refresh_parameter_values_by_entity_class(*db_map_data*)

Refreshes cached parameter values in cascade when updating entity classes.

Parameters
db_map_data (*dict*) – lists of updated items keyed by DiffDatabaseMapping

_cascade_refresh_parameter_values_by_entity(*db_map_data*)

Refreshes cached parameter values in cascade when updating entities.

Parameters
db_map_data (*dict*) – lists of updated items keyed by DiffDatabaseMapping

_cascade_refresh_parameter_values_by_alternative(*db_map_data*)

Refreshes cached parameter values in cascade when updating alternatives.

Parameters
db_map_data (*dict*) – lists of updated items keyed by DiffDatabaseMapping

_cascade_refresh_parameter_values_by_definition(*db_map_data*)

Refreshes cached parameter values in cascade when updating parameter definitions.

Parameters
db_map_data (*dict*) – lists of updated items keyed by DiffDatabaseMapping

_cascade_refresh_features_by_paremeter_definition(*db_map_data*)

Refreshes cached features in cascade when updating parameter definitions.

Parameters
db_map_data (*dict*) – lists of updated items keyed by DiffDatabaseMapping

_cascade_refresh_features_by_parameter_value_list(*db_map_data*)

Refreshes cached features in cascade when updating parameter value lists.

Parameters
db_map_data (*dict*) – lists of updated items keyed by DiffDatabaseMapping

_cascade_refresh_tool_features_by_feature(*db_map_data*)

Refreshes cached tool features in cascade when updating features.

Parameters
db_map_data (*dict*) – lists of updated items keyed by DiffDatabaseMapping

_cascade_refresh_parameter_values_by_list_value(*db_map_data*)

Refreshes cached parameter values in cascade when updating list values.

Parameters
db_map_data (*dict*) – lists of updated items keyed by DiffDatabaseMapping

_cascade_refresh_parameter_definitions_by_list_value(*db_map_data*)

Refreshes cached parameter definitions in cascade when updating list values.

Parameters
db_map_data (*dict*) – lists of updated items keyed by DiffDatabaseMapping

duplicate_scenario(*scen_data*, *dup_name*, *db_map*)

duplicate_object(*object_data*, *orig_name*, *dup_name*, *db_maps*)

_get_data_for_export(*db_map_item_ids*)

export_data(*caller*, *db_map_item_ids*, *file_path*, *file_filter*)

_is_url_available(*url*, *logger*)

export_to_sqlite(*file_path*, *data_for_export*, *caller*)

Exports given data into SQLite file.

export_to_json(*file_path*, *data_for_export*, *caller*)

Exports given data into JSON file.

export_to_excel(*file_path*, *data_for_export*, *caller*)

Exports given data into Excel file.

get_entity_metadata(*db_map*, *entity_id*)

Returns metadata records for given entity.

Parameters

- **db_map** (*DiffDatabaseMapping*) – database mapping
- **entity_id** (*int*) – entity id

Returns
entity metadata records

Return type
list of namedtuple

get_parameter_value_metadata(*db_map*, *parameter_value_id*)

Returns metadata records for given parameter value.

Parameters

- **db_map** (*DiffDatabaseMapping*) – database mapping
- **parameter_value_id** (*int*) – parameter value id

Returns
parameter value metadata records

Return type
list of namedtuple

get_items_for_commit(*db_map*, *commit_id*)

static get_all_multi_spine_db_editors()

Yields all instances of MultiSpineDBEditor currently open.

Yields
MultiSpineDBEditor

get_all_spine_db_editors()

Yields all instances of SpineDBEditor currently open.

Yields
SpineDBEditor

_get_existing_spine_db_editor(*db_url_codenames*)

open_db_editor(*db_url_codenames*)

Opens a SpineDBEditor with given urls. Uses an existing MultiSpineDBEditor if any. Also, if the same urls are open in an existing SpineDBEditor, just raises that one instead of creating another.

Parameters
db_url_codenames (*dict*) – mapping url to codename

spinetoolbox.spine_db_parcel

SpineDBParcel class.

authors

M. Marin (KTH)

date

10.5.2020

Module Contents

Classes

SpineDBParcel	A class to create parcels of data from a Spine db.

class spinetoolbox.spine_db_parcel.**SpineDBParcel**(*db_mngr*)

A class to create parcels of data from a Spine db. Mainly intended for the *Export selection* action in the Spine db editor:

- `push` methods push items with everything they need to live in a standalone db.
- `full_push` and `inner_push` methods do something more specific

Initializes the parcel object.

Parameters

db_mngr (SpineDBManager) –

property data

_get_fields(*db_map*, *item_type*, *field*, *ids*)

Returns a list of field values for items of given type, having given ids.

push_object_class_ids(*db_map_ids*)

Pushes object_class ids.

push_relationship_class_ids(*db_map_ids*)

Pushes relationship_class ids.

push_object_ids(*db_map_ids*)

Pushes object ids.

push_relationship_ids(*db_map_ids*)

Pushes relationship ids.

push_parameter_value_list_ids(*db_map_ids*)

Pushes parameter_value_list ids.

push_parameter_definition_ids(*db_map_ids*, *entity_type*)

Pushes parameter_definition ids.

push_parameter_value_ids(*db_map_ids*, *entity_type*)

Pushes parameter_value ids.

push_object_group_ids(*db_map_ids*)

Pushes object group ids.

push_alternative_ids(*db_map_ids*)

Pushes alternative ids.

push_scenario_ids(*db_map_ids*)

Pushes scenario ids.

push_scenario_alternative_ids(*db_map_ids*)

Pushes scenario_alternative ids.

push_feature_ids(*db_map_ids*)

Pushes feature ids.

push_tool_ids(*db_map_ids*)

Pushes tool ids.

push_tool_feature_ids(*db_map_ids*)

Pushes tool_feature ids.

push_tool_feature_method_ids(*db_map_ids*)

Pushes tool_feature_method ids.

full_push_object_class_ids(*db_map_ids*)

Pushes parameter definitions associated with given object classes. This essentially full_pushes the object classes and their parameter definitions.

full_push_relationship_class_ids(*db_map_ids*)

Pushes parameter definitions associated with given relationship classes. This essentially full_pushes the relationships classes, their parameter definitions, and their member object classes.

full_push_object_ids(*db_map_ids*)

Pushes parameter values associated with objects and with any relationships involving those objects. This essentially full_pushes objects, their relationships, all the parameter values, and all the necessary classes, definitions, and lists.

full_push_relationship_ids(*db_map_ids*)

Pushes parameter values associated with relationships. This essentially full_pushes relationships, their parameter values, and all the necessary classes, definitions, and lists.

full_push_scenario_ids(*db_map_ids*)

inner_push_object_ids(*db_map_ids*)

Pushes object ids, cascading relationship ids, and the associated parameter values, but not any entity classes or parameter definitions. Mainly intended for the *Duplicate object* action.

inner_push_relationship_ids(*db_map_ids*)

Pushes relationship ids, and the associated parameter values, but not any entity classes or parameter definitions.

inner_push_parameter_value_ids(*db_map_ids*, *entity_type*)

Pushes parameter_value ids.

_update_ids(*db_map_ids*, *key*)

Updates ids for given database item.

Parameters

- **db_map_ids** (*dict*) – mapping from `DatabaseMappingBase` to ids or `Asterisk`
- **key** (*str*) – the key

_setdefault(*db_map*)

Adds new id sets for given `db_map` or returns existing ones.

Parameters
db_map (*DatabaseMappingBase*) – a database map

Returns
mapping from item name to set of ids

Return type
dict

spinetoolbox.spine_db_signaller

Spine DB Signaller class.

authors

M. Marin (KTH)

date
31.10.2019

Module Contents

Classes

SpineDBSignaller	Handles signals from DB manager and channels them to listeners.

class spinetoolbox.spine_db_signaller.**SpineDBSignaller**(*db_mngr*)

Bases: `PySide2.QtCore.QObject`

Handles signals from DB manager and channels them to listeners.

Initializes the signaler object.

Parameters
db_mngr (`SpineDBManager`) –

pause(**db_maps*)

Defers notifications from db_maps until `resume` is called.

resume(**db_maps*)

Performs deferred notifications.

_defer_notification(*db_map*, *data*, *method*)

Defer notification if db_map is paused, and returns True. Otherwise returns False.

add_db_map_listener(*db_map*, *listener*)

Adds listener for given db_map.

remove_db_map_listener(*db_map*, *listener*)

Removes db_map from the maps listener listens to.

db_map_listeners(*db_map*)

db_map_editors(*db_map*)

Creates a set of given database map's editors.

Editors are listeners that are interested whether the database is dirty. They are expected to implement `is_db_map_editor()` which returns a bool.

Parameters
db_map (*DiffDatabaseMapping*) – database map

Returns
database mapping's editors

Return type
set

connect_signals()

Connects signals.

receive_scenarios_added(*db_map_data*)

receive_alternatives_added(*db_map_data*)

receive_object_classes_added(*db_map_data*)

receive_objects_added(*db_map_data*)

receive_relationship_classes_added(*db_map_data*)

receive_relationships_added(*db_map_data*)

receive_entity_groups_added(*db_map_data*)

receive_parameter_definitions_added(*db_map_data*)

receive_parameter_values_added(*db_map_data*)

receive_parameter_value_lists_added(*db_map_data*)

receive_list_values_added(*db_map_data*)

receive_features_added(*db_map_data*)

receive_tools_added(*db_map_data*)

receive_tool_features_added(*db_map_data*)

receive_tool_feature_methods_added(*db_map_data*)

receive_metadata_added(*db_map_data*)

receive_entity_metadata_added(*db_map_data*)

receive_parameter_value_metadata_added(*db_map_data*)

receive_scenarios_updated(*db_map_data*)

receive_alternatives_updated(*db_map_data*)

receive_object_classes_updated(*db_map_data*)

receive_objects_updated(*db_map_data*)

receive_relationship_classes_updated(*db_map_data*)

receive_relationships_updated(*db_map_data*)

receive_parameter_definitions_updated(*db_map_data*)

receive_parameter_values_updated(*db_map_data*)

receive_parameter_value_lists_updated(*db_map_data*)

receive_list_values_updated(*db_map_data*)

receive_features_updated(*db_map_data*)

receive_tools_updated(*db_map_data*)

receive_tool_features_updated(*db_map_data*)

receive_tool_feature_methods_updated(*db_map_data*)

receive_metadata_updated(*db_map_data*)

receive_entity_metadata_updated(*db_map_data*)

receive_parameter_value_metadata_updated(*db_map_data*)

receive_scenarios_removed(*db_map_data*)

receive_alternatives_removed(*db_map_data*)

receive_object_classes_removed(*db_map_data*)

receive_objects_removed(*db_map_data*)

receive_relationship_classes_removed(*db_map_data*)

receive_relationships_removed(*db_map_data*)

receive_entity_groups_removed(*db_map_data*)

receive_parameter_definitions_removed(*db_map_data*)

receive_parameter_values_removed(*db_map_data*)

receive_parameter_value_lists_removed(*db_map_data*)

receive_list_values_removed(*db_map_data*)

receive_features_removed(*db_map_data*)

receive_tools_removed(*db_map_data*)

receive_tool_features_removed(*db_map_data*)

receive_tool_feature_methods_removed(*db_map_data*)

receive_metadata_removed(*db_map_data*)

receive_entity_metadata_removed(*db_map_data*)

receive_parameter_value_metadata_removed(*db_map_data*)

receive_error_msg(*db_map_error_log*)

static _shared_db_map_data(*db_map_data*, *db_maps*)

_call_in_listeners(*callback*, *db_map_data*)

receive_session_refreshed(*db_maps*)

receive_session_committed(*db_maps*, *cookie*)

receive_session_rolled_back(*db_maps*)

spinetoolbox.spine_db_worker

The SpineDBWorker class

authors

P. Vennström (VTT) and M. Marin (KTH)

date

2.10.2019

Module Contents

Classes

_Event	Generic enumeration.
SpineDBWorker	Does all the communication with a certain DB for SpineDBManager, in a non-GUI thread.

Functions

_db_map_lock(func)	A wrapper for SpineDBWorker that locks the database for the duration of the wrapped method.
_make_iterator(query)	Runs the given query and yields results by chunks of given size.

Attributes

_CHUNK_SIZE

class spinetoolbox.spine_db_worker.**_Event**

Bases: enum.Enum

Generic enumeration.

Derive from this class to define new enumerations.

FETCH

FETCH_STATUS_CHANGE

ADD_OR_UPDATE_ITEMS

READD_ITEMS

REMOVE_ITEMS

COMMIT_SESSION

ROLLBACK_SESSION

spinetoolbox.spine_db_worker.**_CHUNK_SIZE = 1000**

spinetoolbox.spine_db_worker.**_db_map_lock**(*func*)

A wrapper for SpineDBWorker that locks the database for the duration of the wrapped method.

In case the locking fails, the wrapped method will not be invoked.

Parameters
func (*Callable*) – method to wrap

class spinetoolbox.spine_db_worker.**SpineDBWorker**(*db_mngr*, *db_url*)

Bases: PySide2.QtCore.QObject

Does all the communication with a certain DB for SpineDBManager, in a non-GUI thread.

_something_happened

clean_up()

_handle_something_happened(*event*, *args*)

query(*sq_name*)

For tests.

_query(*sq_name*)

get_db_map(**args*, ***kwargs*)

_get_db_map(**args*, ***kwargs*)

reset_queries()

Resets queries and clears caches.

_reset_fetching_if_required(*parent*)

Sets fetch parent's token or resets the parent if fetch tokens don't match.

Parameters
parent (FetchParent) – fetch parent

can_fetch_more(*parent*)

Returns whether more data can be fetches for parent.

Parameters
parent (FetchParent) – fetch parent

Returns
True if more data is available, False otherwise

Return type
bool

_init_query(*parent*)

Initializes query for parent.

Parameters
parent (FetchParent) – fetch parent

static _fetch_status_change_event(*parent*)

_setdefault_query(*parent*)

Creates a query for parent. Stores both the query and whether it has elements.

Parameters
parent (FetchParent) – fetch parent

_query_has_elements(*parent*)

Checks whether query has something to return.

Parameters
parent (FetchParent) – fetch parent

Returns
True if query will give records, False otherwise

Return type
bool

static _setdefault_query_key(*parent*)

Returns parent's query key or creates and sets a new one if it doesn't exist.

Parameters
parent (FetchParent) – fetch parent

Returns
query key

Return type
str

fetch_more(*parent*)

Fetches items from the database.

Parameters
parent (FetchParent) – fetch parent

_fetch_more(*parent*)

_fetch_event(*parent*, *chunk*)

fetch_all(*item_types=None*, *only_descendants=False*, *include_ancestors=False*)

_fetch_all(*item_types*)

_make_query_for_parent(*parent*)

Makes a database query for given item type.

Parameters
parent (*object*) – the object that requests the fetching

Returns
database query

Return type
Query

_make_query_for_item_type(*item_type*)

_get_iterator(*parent*)

_populate_commit_cache(*item_type*, *items*)

close_db_map()

_close_db_map()

get_entity_metadata(*entity_id*)

Queries metadata records for a single entity synchronously.

Parameters
entity_id (*int*) – entity id

Returns
entity metadata records

Return type
list of namedtuple

_get_entity_metadata(*entity_id*)

Queries metadata records for a single entity.

Parameters
entity_id (*int*) – entity id

Returns
entity metadata records

Return type
list of namedtuple

get_parameter_value_metadata(*parameter_value_id*)

Queries metadata records for a single parameter value synchronously.

Parameters
parameter_value_id (*int*) – parameter value id

Returns
parameter value metadata records

Return type
list of namedtuple

_get_parameter_value_metadata(*parameter_value_id*)

Queries metadata records for a single parameter value.

Parameters
parameter_value_id (*int*) – parameter value id

Returns
parameter value metadata records

Return type
list of namedtuple

add_or_update_items(*items*, *method_name*, *item_type*, *signal_name*, *check*, *cache*)

Adds or updates items in db.

Parameters

- **items** (*dict*) – lists of items to add or update
- **method_name** (*str*) – attribute of DiffDatabaseMapping to call for performing the operation
- **item_type** (*str*) – item type
- **signal_name** (*str*) – signal attribute of SpineDBManager to emit if successful
- **check** (*bool*) – Whether or not to check integrity
- **cache** (*dict*) – Cache

_add_or_update_items(*items*, *method_name*, *item_type*, *signal_name*, *check*, *cache*)

_add_or_update_items_event(*items*, *errors*, *signal_name*)

readd_items(*items*, *method_name*, *item_type*, *signal_name*, *cache*)

Adds or updates items in db.

Parameters

- **items** (*dict*) – lists of items to add or update
- **method_name** (*str*) – attribute of DiffDatabaseMapping to call for performing the operation
- **item_type** (*str*) – item type
- **signal_name** (*str*) – signal attribute of SpineDBManager to emit if successful

_readd_items(*items*, *method_name*, *item_type*, *signal_name*, *cache*)

_readd_items_event(*items*, *signal_name*)

remove_items(*ids_per_type*)

Removes items from database.

Parameters
ids_per_type (*dict*) – lists of items to remove keyed by item type (str)

_remove_items(*ids_per_type*)

_remove_items_event(*ids_per_type*, *errors*)

commit_session(*commit_msg*, *cookie=None*)

Initiates commit session.

Parameters

- **commit_msg** (*str*) – commit message
- **cookie** (*Any*) – a cookie to include in session_committed signal

_commit_session(*commit_msg*, *undo_stack*, *cookie=None*)

Commits session for given database maps.

Parameters

- **commit_msg** (*str*) – commit message
- **undo_stack** (AgedUndoStack) – undo stack that outlive the DB manager
- **cookie** (*Any*) – a cookie to include in session_committed signal

_commit_session_event(*errors*, *undo_stack*, *cookie*)

rollback_session()

Initiates rollback session action for given database maps in the worker thread.

_rollback_session(*undo_stack*)

Rolls back session for given database maps.

Parameters

undo_stack (AgedUndoStack) – undo stack that outlive the DB manager

_rollback_session_event(*errors*, *undo_stack*)

spinetoolbox.spine_db_worker.**_make_iterator**(*query*)

Runs the given query and yields results by chunks of given size.

Parameters

query (*Query*) – the query

Yields

list – chunk of items

spinetoolbox.spine_engine_manager

Contains SpineEngineManagerBase.

authors

M. Marin (KTH), P. Pääkkönen (VTT), P. Savolainen (VTT)

date

14.10.2020

Module Contents

Classes

SpineEngineManagerBase	
LocalSpineEngineManager	
RemoteSpineEngineManager	Responsible for remote project execution.

Functions

make_engine_manager([remote_execution_enabled, job_id])	Returns either a Local or a remote Spine Engine Manager based on settings.

class spinetoolbox.spine_engine_manager.**SpineEngineManagerBase**

abstract run_engine(*engine_data*)

Runs an engine with given data.

Parameters
engine_data (*dict*) – The engine data.

abstract get_engine_event()

Gets next event from a running engine.

Returns
two element tuple: event type identifier string, and event data dictionary

Return type
tuple(str,dict)

abstract stop_engine()

Stops a running engine.

abstract answer_prompt(*item_name*, *accepted*)

Answers prompt.

Parameters

- **item_name** (*str*) – The item that emitted the prompt
- **accepted** (*bool*) – The user's decision.

abstract restart_kernel(*connection_file*)

Restarts the jupyter kernel associated to given connection file.

Parameters
connection_file (*str*) – path of connection file

abstract shutdown_kernel(*connection_file*)

Shuts down the jupyter kernel associated to given connection file.

Parameters
connection_file (*str*) – path of connection file

abstract issue_persistent_command(*persistent_key*, *command*)

Issues a command to a persistent process.

Parameters

- **persistent_key** (*tuple*) – persistent identifier
- **command** (*str*) – command to issue

Returns
stdin, stdout, and stderr messages (dictionaries with two keys: type, and data)

Return type
generator

abstract is_persistent_command_complete(*persistent_key*, *command*)

Checks whether a command is complete.

Parameters

- **key** (*tuple*) – persistent identifier
- **cmd** (*str*) – command to issue

Returns
bool

abstract restart_persistent(*persistent_key*)

Restarts a persistent process.

Parameters
persistent_key (*tuple*) – persistent identifier

Returns
stdout and stderr messages (dictionaries with two keys: type, and data)

Return type
generator

abstract interrupt_persistent(*persistent_key*)

Interrupts a persistent process.

Parameters
persistent_key (*tuple*) – persistent identifier

abstract get_persistent_completions(*persistent_key*, *text*)

Returns a list of auto-completion options from given text.

Parameters

- **persistent_key** (*tuple*) – persistent identifier
- **text** (*str*) – text to complete

Returns
list of str

abstract get_persistent_history_item(*persistent_key*, *text*, *prefix*, *backwards*)

Returns an item from persistent history.

Parameters
persistent_key (*tuple*) – persistent identifier

Returns
history item or empty string if none

Return type
str

class spinetoolbox.spine_engine_manager.**LocalSpineEngineManager**

Bases: *SpineEngineManagerBase*

run_engine(*engine_data*)

Runs an engine with given data.

Parameters
engine_data (*dict*) – The engine data.

get_engine_event()

Gets next event from a running engine.

Returns
two element tuple: event type identifier string, and event data dictionary

Return type
tuple(str,dict)

stop_engine()

Stops a running engine.

answer_prompt(*item_name*, *accepted*)

Answers prompt.

Parameters

- **item_name** (*str*) – The item that emitted the prompt
- **accepted** (*bool*) – The user's decision.

restart_kernel(*connection_file*)

Restarts the jupyter kernel associated to given connection file.

Parameters
connection_file (*str*) – path of connection file

shutdown_kernel(*connection_file*)

Shuts down the jupyter kernel associated to given connection file.

Parameters
connection_file (*str*) – path of connection file

issue_persistent_command(*persistent_key*, *command*)

Issues a command to a persistent process.

Parameters

- **persistent_key** (*tuple*) – persistent identifier
- **command** (*str*) – command to issue

Returns
stdin, stdout, and stderr messages (dictionaries with two keys: type, and data)

Return type
generator

is_persistent_command_complete(*persistent_key*, *command*)

Checks whether a command is complete.

Parameters

- **key** (*tuple*) – persistent identifier
- **cmd** (*str*) – command to issue

Returns
bool

restart_persistent(*persistent_key*)

Restarts a persistent process.

Parameters
persistent_key (*tuple*) – persistent identifier

Returns
stdout and stderr messages (dictionaries with two keys: type, and data)

Return type
generator

interrupt_persistent(*persistent_key*)

Interrupts a persistent process.

Parameters
persistent_key (*tuple*) – persistent identifier

get_persistent_completions(*persistent_key*, *text*)

Returns a list of auto-completion options from given text.

Parameters

- **persistent_key** (*tuple*) – persistent identifier
- **text** (*str*) – text to complete

Returns
list of str

get_persistent_history_item(*persistent_key*, *text*, *prefix*, *backwards*)

Returns an item from persistent history.

Parameters
persistent_key (*tuple*) – persistent identifier

Returns
history item or empty string if none

Return type
str

class spinetoolbox.spine_engine_manager.**RemoteSpineEngineManager**(*job_id=''*)

Bases: *SpineEngineManagerBase*

Responsible for remote project execution.

Initializer.

make_engine_client(*host*, *port*, *security*, *sec_folder*, *ping=True*)

Creates a client for connecting to Spine Engine Server.

run_engine(*engine_data*)

Makes an engine client for communicating with the engine server. Starts a thread for monitoring the DAG execution on server.

Parameters
engine_data (*dict*) – The engine data.

get_engine_event()

Returns the next engine execution event.

clean_up()

Closes EngineClient and joins _runner thread if still active.

stop_engine()

Sends a request to stop execution on Server then waits for _runner thread to end.

_run()

Sends a start execution request to server with the job Id. Sets up a subscribe socket according to the publish port received from server. Passes received events to SpineEngineWorker for processing. After execution has finished, downloads new files from server.

abstract answer_prompt(*item_name*, *accepted*)

See base class.

restart_kernel(*connection_file*)

See base class.

shutdown_kernel(*connection_file*)

See base class.

is_persistent_command_complete(*persistent_key*, *command*)

Checks whether a command is complete.

Parameters

- **key** (*tuple*) – persistent identifier
- **cmd** (*str*) – command to issue

Returns
bool

issue_persistent_command(*persistent_key*, *command*)

Issues a command to a persistent process.

Parameters

- **persistent_key** (*tuple*) – persistent identifier
- **command** (*str*) – command to issue

Returns
stdin, stdout, and stderr messages (dictionaries with two keys: type, and data)

Return type
generator

restart_persistent(*persistent_key*)

See base class.

interrupt_persistent(*persistent_key*)

See base class.

get_persistent_completions(*persistent_key*, *text*)

See base class.

get_persistent_history_item(*persistent_key*, *text*, *prefix*, *backwards*)

Returns an item from persistent history.

Parameters

persistent_key (*tuple*) – persistent identifier

Returns

history item or empty string if none

Return type

str

spinetoolbox.spine_engine_manager.**make_engine_manager**(*remote_execution_enabled=False*, *job_id=''*)

Returns either a Local or a remote Spine Engine Manager based on settings.

Parameters

- **remote_execution_enabled** (*bool*) – True returns a local Spine Engine Manager instance,
- **instance** (*False returns a remote Spine Engine Manager*) –
- **job_id** (*str*) – Server execution job Id

spinetoolbox.spine_engine_worker

Contains SpineEngineWorker. :authors: M. Marin (KTH) :date: 14.10.2020

Module Contents

Classes

SpineEngineWorker

param engine_data
engine data

Functions

_handle_dag_execution_started(project_items)	
_handle_node_execution_started(item, direction)	
_handle_node_execution_finished(item, direction, ...)	
_handle_event_message_arrived(item, filter_id, ...)	
_handle_process_message_arrived(item, filter_id, ...)	
_handle_prompt_arrived(prompt, engine_mngr)	
_handle_flash_arrived(connection)	
_mark_all_items_failed(items)	Fails all project items.

spinetoolbox.spine_engine_worker.**_handle_dag_execution_started**(*project_items*)

spinetoolbox.spine_engine_worker.**_handle_node_execution_started**(*item*, *direction*)

spinetoolbox.spine_engine_worker.**_handle_node_execution_finished**(*item*, *direction*, *item_state*)

spinetoolbox.spine_engine_worker.**_handle_event_message_arrived**(*item*, *filter_id*, *msg_type*, *msg_text*)

spinetoolbox.spine_engine_worker.**_handle_process_message_arrived**(*item*, *filter_id*, *msg_type*, *msg_text*)

spinetoolbox.spine_engine_worker.**_handle_prompt_arrived**(*prompt*, *engine_mngr*)

spinetoolbox.spine_engine_worker.**_handle_flash_arrived**(*connection*)

spinetoolbox.spine_engine_worker.**_mark_all_items_failed**(*items*)

Fails all project items.

Parameters

items (*list of* ProjectItem) – project items

class spinetoolbox.spine_engine_worker.**SpineEngineWorker**(*engine_data*, *dag*, *dag_identifier*, *project_items*, *connections*, *logger*, *job_id*)

Bases: PySide2.QtCore.QObject

Parameters

- **engine_data** (*dict*) – engine data
- **dag** (*DirectedGraphHandler*) –
- **dag_identifier** (*str*) –
- **project_items** (*dict*) – mapping from project item name to ProjectItem
- **connections** (*dict*) – mapping from jump name to LoggingConnection or LoggingJump

- **logger** (LoggerInterface) – a logger
- **job_id** – Job Id for remote execution

property engine_data

Engine data dictionary.

finished

_dag_execution_started

_node_execution_started

_node_execution_finished

_event_message_arrived

_process_message_arrived

_prompt_arrived

_flash_arrived

_all_items_failed

get_engine_data()

Returns the engine data. Together with `self.set_engine_data()` it can be used to modify the workflow after it's initially created. We use it at the moment for creating Julia sysimages.

Returns
dict

set_engine_data(*engine_data*)

Sets the engine data.

Parameters
engine_data (*dict*) – New data

_handle_event_message_arrived_silent(*item*, *filter_id*, *msg_type*, *msg_text*)

_handle_process_message_arrived_silent(*item*, *filter_id*, *msg_type*, *msg_text*)

stop_engine()

engine_final_state()

thread()

_connect_log_signals(*silent*)

start(*silent=False*)

Connects log signals.

Parameters
silent (*bool, optional*) – If True, log messages are not forwarded to the loggers but saved in internal dicts.

do_work()

Does the work and emits finished when done.

_process_event(*event_type*, *data*)

_handle_prompt(*prompt*)

_handle_flash(*flash*)

_handle_standard_execution_msg(*msg*)

_handle_persistent_execution_msg(*msg*)

_handle_kernel_execution_msg(*msg*)

_handle_process_msg(*data*)

_do_handle_process_msg(*item_name*, *filter_id*, *msg_type*, *msg_text*)

_handle_event_msg(*data*)

_do_handle_event_msg(*item_name*, *filter_id*, *msg_type*, *msg_text*)

_handle_node_execution_started(*data*)

_do_handle_node_execution_started(*item_name*, *direction*)

Starts item icon animation when executing forward.

_handle_node_execution_finished(*data*)

_do_handle_node_execution_finished(*item_name*, *direction*, *state*, *item_state*)

_handle_server_status_msg(*data*)

clean_up()

spinetoolbox.ui_main

Contains ToolboxUI class.

author

P. Savolainen (VTT)

date

14.12.2017

Module Contents

Classes

ToolboxUI	Class for application main GUI functions.

class spinetoolbox.ui_main.**ToolboxUI**

Bases: PySide2.QtWidgets.QMainWindow

Class for application main GUI functions.

Initializes application and main window.

msg

msg_success

msg_error

msg_warning

msg_proc

msg_proc_error

information_box

error_box

jupyter_console_requested

persistent_console_requested

eventFilter(*obj*, *ev*)

_setup_properties_title()

connect_signals()

Connect signals.

_open_active_item_dir(*_checked=False*)

static set_error_mode()

Sets Windows error mode to show all error dialog boxes from subprocesses.

See https://docs.microsoft.com/en-us/windows/win32/api/errhandlingapi/nf-errhandlingapi-seterrormode for documentation.

_update_qsettings()

Updates obsolete settings.

_update_execute_enabled()

_update_execute_selected_enabled()

update_window_modified(*clean*)

Updates window modified status and save actions depending on the state of the undo stack.

parse_project_item_modules()

Collects data from project item factories.

set_work_directory(*new_work_dir=None*)

Creates a work directory if it does not exist or changes the current work directory to given.

Parameters

new_work_dir(*str, optional*) – If given, changes the work directory to given and creates the directory if it does not exist.

project()

Returns current project or None if no project open.

Returns

current project or None

Return type

SpineToolboxProject

qsettings()

Returns application preferences object.

item_specification_factories()

Returns project item specification factories.

Returns
specification factories

Return type
list of ProjectItemSpecificationFactory

update_window_title()

Updates main window title.

init_project(*project_dir*)

Initializes project at application start-up.

Opens the last project that was open when app was closed (if enabled in Settings) or starts the app without a project.

Parameters
project_dir (*str*) – project directory

new_project()

Opens a file dialog where user can select a directory where a project is created. Pops up a question box if selected directory is not empty or if it already contains a Spine Toolbox project. Initial project name is the directory name.

create_project(*name*, *description*, *location*)

Creates new project and sets it active.

Parameters

- **name** (*str*) – Project name
- **description** (*str*) – Project description
- **location** (*str*) – Path to project directory

open_project(*load_dir=None*)

Opens project from a selected or given directory.

Parameters
load_dir (*str, optional*) – Path to project base directory. If default value is used, a file explorer dialog is opened where the user can select the project to open.

Returns
True when opening the project succeeded, False otherwise

Return type
bool

restore_project(*project_dir*, *ask_confirmation=True*)

Initializes UI, Creates project, models, connections, etc., when opening a project.

Parameters

- **project_dir** (*str*) – Project directory
- **ask_confirmation** (*bool*) – True closes the previous project with a confirmation box if user has enabled this

Returns
True when restoring project succeeded, False otherwise

Return type
bool

`_toolbars()`

Yields all toolbars in the window.

`_disable_project_actions()`

Disables all project-related actions, except New project, Open project and Open recent. Called in the constructor and when closing a project.

`_enable_project_actions()`

Enables all project-related actions. Called when a new project is created and when a project is opened.

`refresh_toolbars()`

Set toolbars' color using highest possible contrast.

`show_recent_projects_menu()`

Updates and sets up the recent projects menu to File-Open recent menu item.

`save_project()`

Saves project.

`save_project_as()`

Asks user for a new project directory and duplicates the current project there. The name of the duplicated project will be the new directory name. The duplicated project is activated.

`close_project(ask_confirmation=True)`

Closes the current project.

Returns
True when no project open or when it's closed successfully, False otherwise.

Return type
bool

`rename_project(_=False)`

Opens a dialog where the user can enter a new name for the project.

`_update_project_name(new_name)`

Updates window title and recent projects.

Parameters
new_name (*str*) – project's new name

`init_project_item_model()`

Initializes project item model. Create root and category items and add them to the model.

`init_specification_model()`

Initializes specification model.

`make_item_properties_uis()`

`_make_properties_tab(properties_ui)`

add_project_items(*items_dict*, *silent=False*)

Pushes an AddProjectItemsCommand to the undo stack.

Parameters

- **items_dict** (*dict*) – mapping from item name to item dictionary
- **silent** (*bool*) – if True, suppress log messages

supports_specifications(*item_type*)

Returns True if given project item type supports specifications.

Returns
True if item supports specifications, False otherwise

Return type
bool

restore_ui()

Restore UI state from previous session.

clear_ui()

Clean UI to make room for a new or opened project.

undo_critical_commands()

Undoes critical commands in the undo stack.

Returns
False if any critical commands aren't successfully undone

Return type
Bool

overwrite_check(*project_dir*)

Checks if given directory is a project directory and/or empty And asks the user what to do in that case.

Parameters
project_dir (*str*) – Abs. path to a directory

Returns
True if user wants to overwrite an existing project or if the directory is not empty and the user wants to make it into a Spine Toolbox project directory anyway. False if user cancels the action.

Return type
bool

item_selection_changed(*selected*, *deselected*)

Synchronizes selection with scene. The scene handles item/link de/activation.

refresh_active_elements(*active_project_item*, *active_link_item*, *selected_item_names*)

_activate_properties_tab()

_set_active_project_item(*active_project_item*)

Parameters
active_project_item (*ProjectItemBase or NoneType*) –

_set_active_link_item(*active_link_item*)

Sets active link and connects to corresponding properties widget.

Parameters
active_link_item (LoggingConnection *or* LoggingJump, *optional*) –

activate_no_selection_tab()

Shows 'No Selection' tab.

activate_item_tab()

Shows active project item properties tab according to item type.

activate_link_tab()

Shows link properties tab.

update_properties_ui()

_get_active_properties_widget()

add_specification(*specification*)

Pushes an AddSpecificationCommand to undo stack.

import_specification()

Opens a file dialog where the user can select an existing specification definition file (.json). If file is valid, pushes AddSpecificationCommand to undo stack.

replace_specification(*name*, *specification*)

Pushes an ReplaceSpecificationCommand to undo stack.

repair_specification(*name*)

Repairs specification if it is broken.

Parameters
name (*str*) – specification's name

prompt_save_location(*title*, *proposed_path*, *file_filter*)

Shows a dialog for the user to select a path to save a file.

Parameters

- **title** (*str*) – dialog window title
- **proposed_path** (*str*) – A proposed location.
- **file_filter** (*str*) – file extension filter

Returns
absolute path or None if dialog was cancelled

Return type
str

_log_specification_saved(*name*, *path*)

Prints a message in the event log, saying that given spec was saved in a certain location, together with a clickable link to change the location.

Parameters

- **name** (*str*) – specification's name
- **path** (*str*) – specification's file path

remove_all_items()

Pushes a RemoveAllProjectItemsCommand to the undo stack.

register_anchor_callback(*url*, *callback*)

Registers a callback for a given anchor in event log, see `open_anchor()`. Used by `ToolFactory.repair_specification()`.

Parameters

- **url** (*str*) – The anchor url
- **callback** (*function*) – A function to call when the anchor is clicked on event log.

open_anchor(*qurl*)

Open file explorer in the directory given in qurl.

Parameters
qurl (*QUrl*) – The url to open

_change_specification_file_location(*name*)

Prompts user for new location for a project item specification.

Delegates saving to project if one is open by pushing a command to the undo stack, otherwise tries to find the specification from the plugin manager.

Parameters
name (*str*) – specification's name

show_specification_context_menu(*ind*, *global_pos*)

Context menu for item specifications.

Parameters

- **ind** (*QModelIndex*) – In the ProjectItemSpecificationModel
- **global_pos** (*QPoint*) – Mouse position

edit_specification(*index*, *item*)

Opens a specification editor widget.

Parameters

- **index** (*QModelIndex*) – Index of the item (from double-click or context menu signal)
- **item** (`ProjectItem`, *optional*) –

remove_specification(*index*)

Removes specification from project.

Parameters
index (*QModelIndex*) – Index of the specification item

open_specification_file(*index*)

Open the specification definition file in the default (.json) text-editor.

Parameters
index (*QModelIndex*) – Index of the item

new_db_editor()

_handle_zoom_minus_pressed()

Slot for handling case when '-' button in menu is pressed.

_handle_zoom_plus_pressed()

Slot for handling case when '+' button in menu is pressed.

_handle_zoom_reset_pressed()

Slot for handling case when 'reset zoom' button in menu is pressed.

add_zoom_action()

Setups zoom widget action in view menu.

restore_dock_widgets()

Dock all floating and or hidden QDockWidgets back to the main window.

_add_actions()

Sets adds actions to the main window.

set_debug_qactions()

Sets shortcuts for QActions that may be needed in debugging.

add_menu_actions()

Adds extra actions to Edit and View menu.

toggle_properties_tabbar_visibility()

Shows or hides the tab bar in properties dock widget. For debugging purposes.

update_datetime()

Returns a boolean, which determines whether date and time is prepended to every Event Log message.

add_message(*msg*)

Append regular message to Event Log.

Parameters

msg (*str*) – String written to QTextBrowser

add_success_message(*msg*)

Append message with green text color to Event Log.

Parameters

msg (*str*) – String written to QTextBrowser

add_error_message(*msg*)

Append message with red color to Event Log.

Parameters

msg (*str*) – String written to QTextBrowser

add_warning_message(*msg*)

Append message with yellow (golden) color to Event Log.

Parameters

msg (*str*) – String written to QTextBrowser

add_process_message(*msg*)

Writes message from stdout to process output QTextBrowser.

Parameters

msg (*str*) – String written to QTextBrowser

add_process_error_message(*msg*)

Writes message from stderr to process output QTextBrowser.

Parameters
msg (*str*) – String written to QTextBrowser

override_console_and_execution_list()

_override_console()

Sets the jupyter console of the active project item in Jupyter Console and updates title.

_do_override_console(*console*)

_override_execution_list()

Displays executions of the active project item in Executions and updates title.

_restore_original_console()

Sets the Console back to the original.

_set_override_console(*console*)

_refresh_console_execution_list()

Refreshes console executions as the active project item starts new executions.

_select_console_execution(*current*, *_previous*)

Sets the console of the selected execution in Console.

show_add_project_item_form(*item_type*, *x=0*, *y=0*, *spec=''*)

Show add project item widget.

supports_specification(*item_type*)

Returns True if given item type supports specifications.

Parameters
item_type (*str*) – item's type

Returns
True if item supports specifications, False otherwise

Return type
bool

show_specification_form(*item_type*, *specification=None*, *item=None*, ***kwargs*)

Shows specification widget.

Parameters

- **item_type** (*str*) – item's type
- **specification** (*ProjectItemSpecification, optional*) – specification
- **item** (ProjectItem, *optional*) – project item
- ****kwargs** – parameters passed to the specification widget

static get_all_multi_tab_spec_editors(*item_type*)

_get_existing_spec_editor(*item_type*, *specification*, *item*)

show_settings()

Show Settings widget.

show_about()

Show About Spine Toolbox form.

show_user_guide()

Open Spine Toolbox documentation index page in browser.

show_getting_started_guide()

Open Spine Toolbox Getting Started HTML page in browser.

retrieve_project()

Retrieves project from server.

engine_server_settings()

Returns the user given Spine Engine Server settings in a tuple.

show_item_context_menu(*pos*)

Context menu for project items listed in the project QTreeView.

Parameters

pos (*QPoint*) – Mouse position

show_project_or_item_context_menu(*pos*, *index*)

Creates and shows the project item context menu.

Parameters

- **pos** (*QPoint*) – Mouse position
- **index** (*QModelIndex, optional*) – Index of concerned item or None

show_link_context_menu(*pos*, *link*)

Context menu for connection links.

Parameters

- **pos** (*QPoint*) – Mouse position
- **link** (Link(*QGraphicsPathItem*)) – The concerned link

refresh_edit_action_states()

Sets the enabled/disabled state for copy, paste, duplicate, and remove actions in File-Edit menu, project tree view context menu, and in Design View context menus just before the menus are shown to user.

enable_edit_actions()

Enables project item edit actions after a QMenu has been shown. This is needed to enable keyboard shortcuts (e.g. Ctrl-C & del) again.

tear_down_consoles()

Closes the 'base' Python and Julia Consoles if running.

_tasks_before_exit()

Returns a list of tasks to perform before exiting the application.

Possible tasks are:

- *"prompt exit"*: prompt user if quitting is really desired
- *"prompt save"*: prompt user if project should be saved before quitting
- *"save"*: save project before quitting

Returns
a list containing zero or more tasks

`_perform_pre_exit_tasks()`

Prompts user to confirm quitting and saves the project if necessary.

Returns
True if exit should proceed, False if the process was cancelled

`_confirm_exit()`

Confirms exiting from user.

Returns
True if exit should proceed, False if user cancelled

`_confirm_save_and_exit()`

Confirms exit from user and saves the project if requested.

Returns
True if exiting should proceed, False if user cancelled

`remove_path_from_recent_projects(p)`

Removes entry that contains given path from the recent project files list in QSettings.

Parameters
p (*str*) – Full path to a project directory

`update_recent_projects()`

Adds a new entry to QSettings variable that remembers twenty most recent project paths.

`closeEvent(event)`

Method for handling application exit.

Parameters
event (*QCloseEvent*) – PySide2 event

`_serialize_selected_items()`

Serializes selected project items into a dictionary.

The serialization protocol tries to imitate the format in which projects are saved.

Returns
a dict containing serialized version of selected project items

Return type
dict

`_deserialized_item_position_shifts(item_dicts)`

Calculates horizontal and vertical shifts for project items being deserialized.

If the mouse cursor is on the Design view we try to place the items unders the cursor. Otherwise the items will get a small shift so they don't overlap a possible item below. In case the items don't fit the scene rect we clamp their coordinates within it.

Parameters
item_dicts (*dict*) – a dictionary of serialized items being deserialized

Returns
a tuple of (horizontal shift, vertical shift) in scene's coordinates

Return type
tuple

static _set_deserialized_item_position(*item_dict*, *shift_x*, *shift_y*, *scene_rect*)

Moves item's position by shift_x and shift_y while keeping it within the limits of scene_rect.

_deserialize_items(*items_dict*, *duplicate_files=False*)

Deserializes project items from a dictionary and adds them to the current project.

Parameters
items_dict (*dict*) – serialized project items

project_item_to_clipboard()

Copies the selected project items to system's clipboard.

project_item_from_clipboard(*duplicate_files=False*)

Adds project items in system's clipboard to the current project.

Parameters
duplicate_files (*bool*) – Duplicate files boolean

duplicate_project_item(*duplicate_files=False*)

Duplicates the selected project items.

propose_item_name(*prefix*)

Proposes a name for a project item.

The format is *prefix_xx* where *xx* is a counter value [01..99].

Parameters
prefix (*str*) – a prefix for the name

Returns
a name string

Return type
str

_share_item_edit_actions()

Adds generic actions to project tree view and Design View.

_show_message_box(*title*, *message*)

Shows an information message box.

_show_error_box(*title*, *message*)

_connect_project_signals()

Connects signals emitted by project.

_execute_project(*_=False*)

Executes all DAGs in project.

_execute_selection(*_=False*)

Executes selected items.

_stop_execution(*_=False*)

Stops execution in progress.

_set_execution_in_progress()

_unset_execution_in_progress()

set_icon_and_properties_ui(*item_name*)

Adds properties UI to given project item.

Parameters
item_name (*str*) – item's name

project_item_properties_ui(*item_type*)

Returns the properties tab widget's ui.

Parameters
item_type (*str*) – project item's type

Returns
item's properties tab widget

Return type
QWidget

project_item_icon(*item_type*)

_open_project_directory(_)

Opens project's root directory in system's file browser.

_open_project_item_directory(_)

Opens project item's directory in system's file browser.

_remove_selected_items(_)

Pushes commands to remove selected project items and links from project.

_rename_project_item(_)

Renames current project item.

item_category_context_menu()

Creates a context menu for category items.

Returns
category context menu

Return type
QMenu

project_item_context_menu(*additional_actions*)

Creates a context menu for project items.

Parameters
additional_actions (*list of QAction*) – actions to be prepended to the menu

Returns
project item context menu

Return type
QMenu

_start_base_julia_console()

Shows and starts the 'base' Julia Console if not running or activates the window if running.

_start_base_python_console()

Shows and starts the 'base' Python Console if not running or activates the window if running.

destroy_base_python_console()

destroy_julia_console()

_setup_jupyter_console(*item*, *filter_id*, *kernel_name*, *connection_file*, *connection_file_dict*)

Sets up jupyter console, eventually for a filter execution.

Parameters

- **item** (ProjectItem) – Item
- **filter_id** (*str*) – Filter identifier
- **kernel_name** (*str*) – Jupyter kernel name
- **connection_file** (*str*) – Path to connection file
- **connection_file_dict** (*dict*) – Contents of connection file when kernel manager runs on Spine Engine Server

_setup_persistent_console(*item*, *filter_id*, *key*, *language*)

Sets up persistent console, eventually for a filter execution.

Parameters

- **item** (ProjectItem) – Item
- **filter_id** (*str*) – Filter identifier
- **key** (*tuple*) – Key
- **language** (*str*) – Language (e.g. 'python' or 'julia')

add_persistent_stdin(*item*, *filter_id*, *data*)

add_persistent_stdout(*item*, *filter_id*, *data*)

add_persistent_stderr(*item*, *filter_id*, *data*)

_get_console(*item*, *filter_id*)

_make_jupyter_console(*item*, *kernel_name*, *connection_file*)

Creates a new JupyterConsoleWidget for given connection file if none exists yet, and returns it.

Parameters

- **item** (ProjectItem) – Item that owns the console
- **kernel_name** (*str*) – Name of the kernel
- **connection_file** (*str*) – Path of kernel connection file

Returns

JupyterConsoleWidget

_make_persistent_console(*item*, *key*, *language*)

Creates a new PersistentConsoleWidget for given process key.

Parameters

- **item** (ProjectItem) – Item that owns the console
- **key** (*tuple*) – persistent process key in spine engine
- **language** (*str*) – for syntax highlighting and prompting, etc.

Returns

PersistentConsoleWidget

_shutdown_engine_kernels()

Shuts down all kernels managed by Spine Engine.

_close_item_consoles()

restore_and_activate()

static _make_log_entry_title(*title*)

start_execution(*timestamp*)

Starts execution.

Parameters

timestamp (*str*) – time stamp

add_log_message(*item_name*, *filter_id*, *message*)

Adds a message to an item's execution log.

Parameters

- **item_name** (*str*) – item name
- **filter_id** (*str*) – filter identifier
- **message** (*str*) – formatted message

spinetoolbox.version

Version info for Spine Toolbox package. Inspired by python sys.version and sys.version_info.

author

P. Savolainen (VTT)

date

8.1.2020

Module Contents

Classes

VersionInfo	A class for a named tuple containing the five components of the version number: major, minor,

Attributes

major
minor
micro
releaselevel
serial
__version_info__
__version__

class spinetoolbox.version.**VersionInfo**

Bases: NamedTuple

A class for a named tuple containing the five components of the version number: major, minor, micro, releaselevel, and serial. All values except releaselevel are integers; the release level is 'dev', 'alpha', 'beta', 'candidate', or 'final'.

major :int

minor :int

micro :int

releaselevel :str

serial :int

__str__() → str

Create a version string following PEP 440

spinetoolbox.version.**major = 0**

spinetoolbox.version.**minor = 6**

spinetoolbox.version.**micro = 13**

spinetoolbox.version.**releaselevel = dev**

spinetoolbox.version.**serial = 0**

spinetoolbox.version.**__version_info__**

spinetoolbox.version.**__version__**

20.1.3 Package Contents

spinetoolbox.**__version__**

spinetoolbox.**__version_info__**

CHAPTER

TWENTYONE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[CB14] Chris Beams. 2014. 'How to Write a Git Commit Message.' https://chris.beams.io/posts/git-commit/

[JF18] Jeff Forcier. 2018. 'Contributing to Open Source Projects.' https://contribution-guide-org.readthedocs.io/

PYTHON MODULE INDEX

S

INDEX

Symbols

B

C

D

E

F

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z