Spine Toolbox Documentation

Release 0.3

Pekka Savolainen, Manuel Marin, Erkka Rinne

Sep 13, 2019

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.

You can either start reading from the first page onwards or go straight to the *Getting Started* section to get you started quickly. If you need help 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 1

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. The following topics are touched (although not exhaustively covered):

1.1 Spine Toolbox Interface

The central element in Spine Toolbox's interface is the *Design View*, where you can visualize and manipulate your project in a pictorial way. Alongside *Design view* there are a few 'dock widgets' that provide additional functionality:

- *Project* provides a more concise view of your project, including:
  - *Items* currently in the project, grouped by category: Data Stores, Data Connections, Tools, Views, and Data Interfaces.
  - *Tool templates* available in the project.
- *Properties* provides an interface to interact with the currently selected project item.

- *Event Log* shows relevant messages about every performed action.
- *Process Log* shows the output of executed Tools.
- *Julia console* provides an interface to interact with the Julia programming language, and also allows Spine Toolbox to execute Julia Tools.
- *Python console* provides an interface to interact with the Python programming language, and also allows Spine Toolbox to execute Python Tools.

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, view, 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:

A) From the application main menu, select **File -> New project...**

B) Press *Ctrl+N*.

The *New Project* form will show up. Type 'hello world' in the name field —we will leave the description empty this time— and click **Ok**.

Congratulations, you have created a new project.

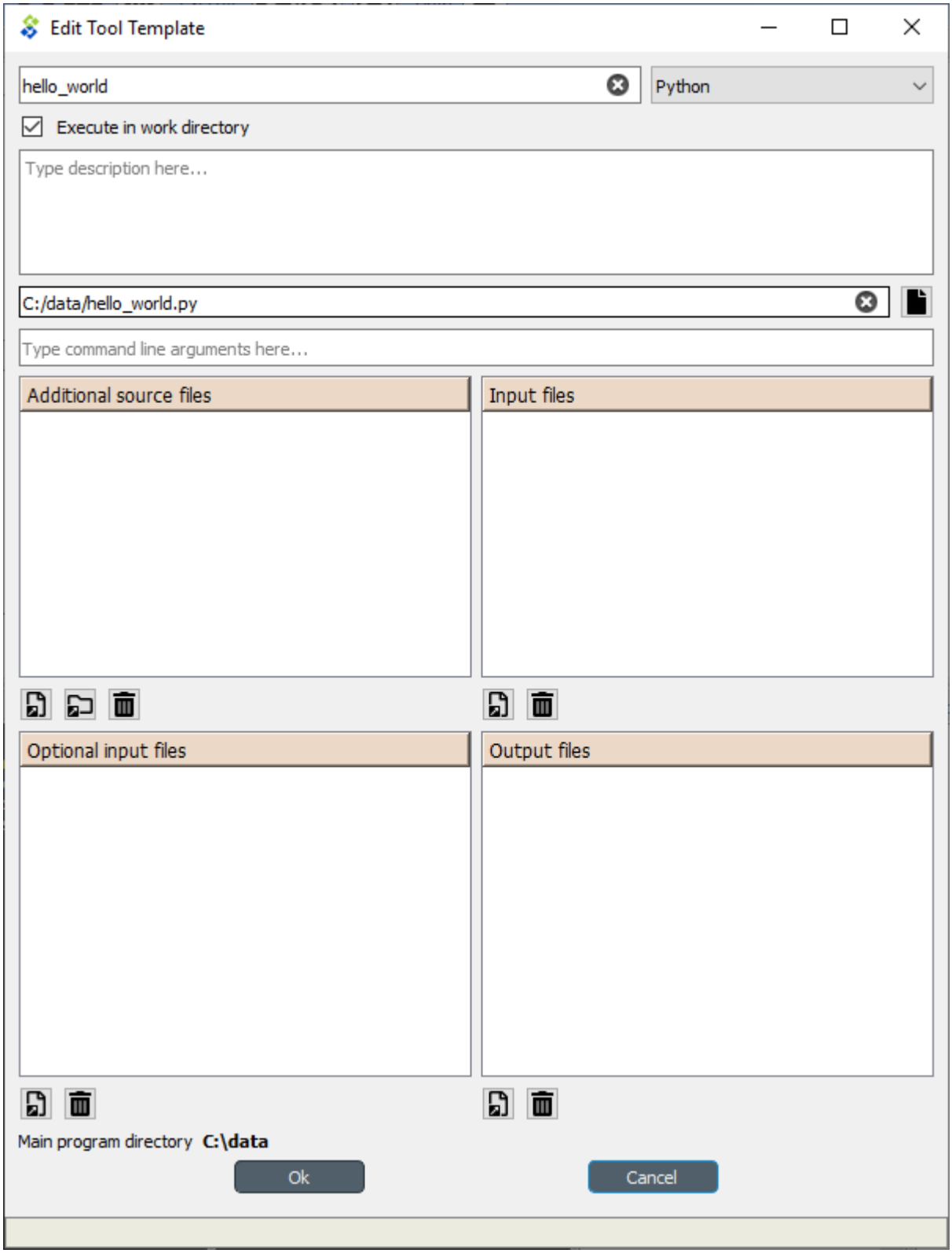
1.3 Creating a Tool template

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

In the *Project* dock widget, click on the 'add tool template button' () just below the *Tool templates* list, and select **New** from the popup menu. The *Edit Tool Template* form will appear. Follow the instructions below to create a minimal Tool template:

- Type 'hello_world' in the *Type name here...* field.
- Select 'Python' from the *Select type...* dropdown list,
- Click on the button right next to the field that reads *Add main program file here...*, and select the option **Make new main program** from the popup menu.
- A file browser dialog should open. Name the file *hello_world.py* and save it in a folder of your choice.

After this, the *Edit Tool Template* form should be looking similar to this:



Click **Ok** at the bottom of the form. A new system dialog will appear, allowing you to select a file name and location to save the Tool template we've just created. Don't change the default file name, which should be *hello_world.json*.

Just select a folder from your system (it can be the same where you saved the main program file) and click **Save**.

Now you should see the new tool template in the *Project* widget, *Tool templates* list.

Tip: Saving the Tool template into a file allows you to add and use the same Tool template in another project. To do this, you just need to click on the add tool button (), select **Add existing...** from the popup menu, and then select the tool template file from your system.

Congratulations, you have just created your first Tool template.

However, the main program file *hello_world.py* was created empty, so for the moment this Tool template does absolutely nothing. To change that, we need to add instructions to that program file so it actually does something when executed.

Right click on the 'hello_world' item in the *Tool templates* list and select **Edit main program file...** from the context menu. This will open the file *hello_world.py* in your default editor.

Enter the following into the file's content:

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

Save the file.

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

1.4 Adding a Tool item to the project

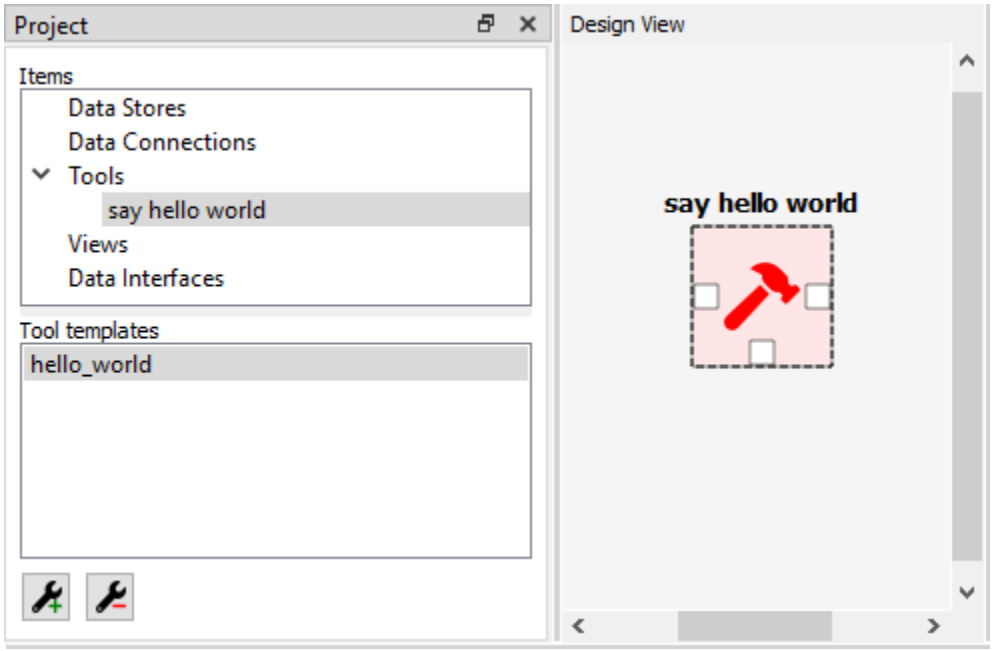
Note: The **Tool** item is used to run Tool templates available in the project.

Let's add a Tool item to our project, so that we're able to run the Tool template we created above. To add a Tool item please do one of the following:

A) From the application main menu, select **Edit -> Add Tool**.

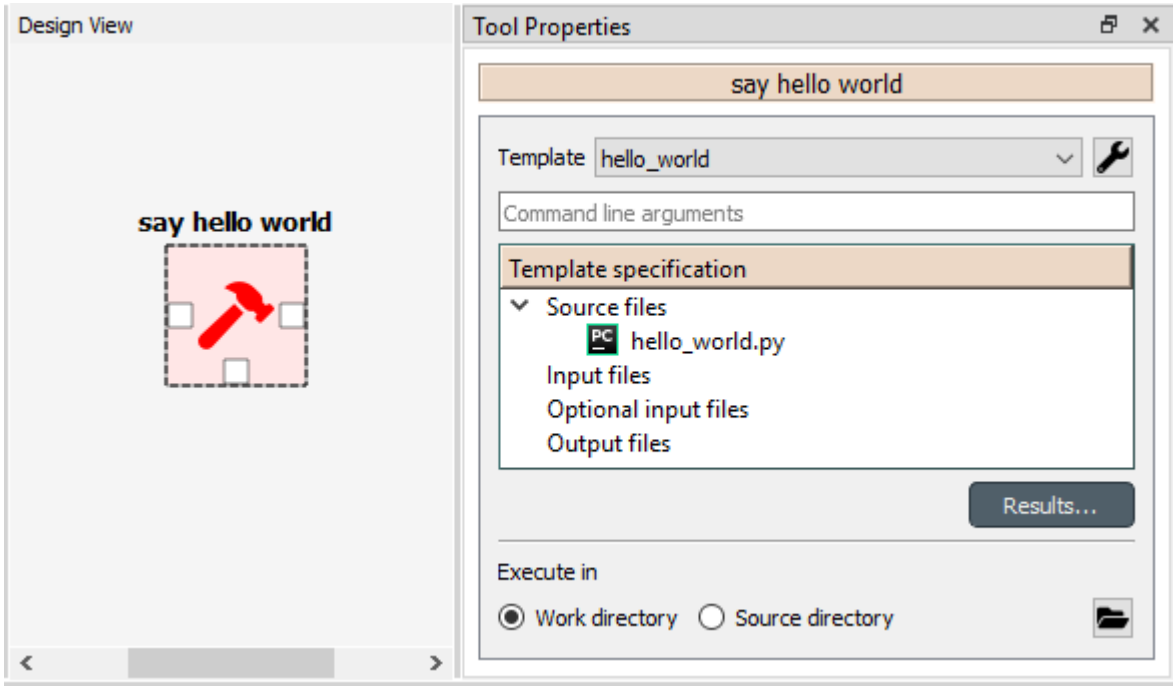
B) Drag-and-drop the Tool icon () from the *Drag & Drop Icon* toolbar onto the *Design View*.

The *Add Tool* form will popup. Type 'say hello world' in the name field, 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:



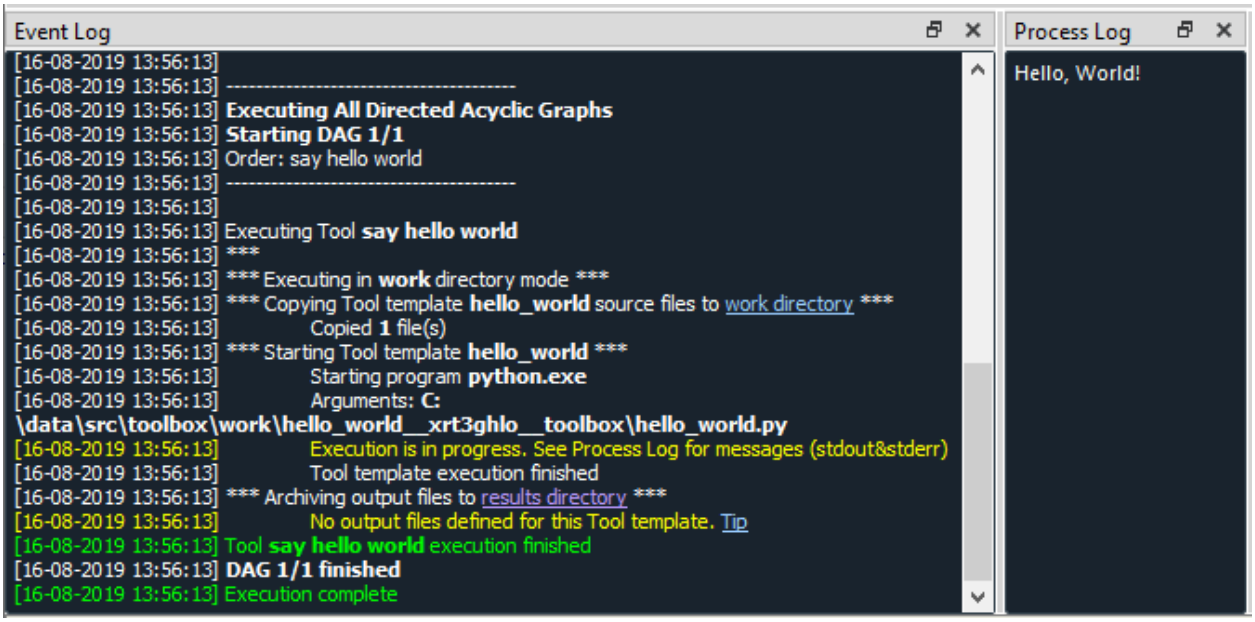
1.5 Executing a Tool

As long as the 'say hello world' Tool item is selected, you will be able to see its *Properties* on the right part of the window, looking similar to this:



Press *execute project* button on the toolbar. This will execute the Tool template 'hello world', which in turn will run the main program file *hello_world.py* in a dedicated process.

You can see more details about execution in the *Event Log*. Once it's finished, you will see its output in the *Process Log* or in the *Python Console* depending on your settings (See *Settings*).



Congratulations, you just ran your first Spine Toolbox project.

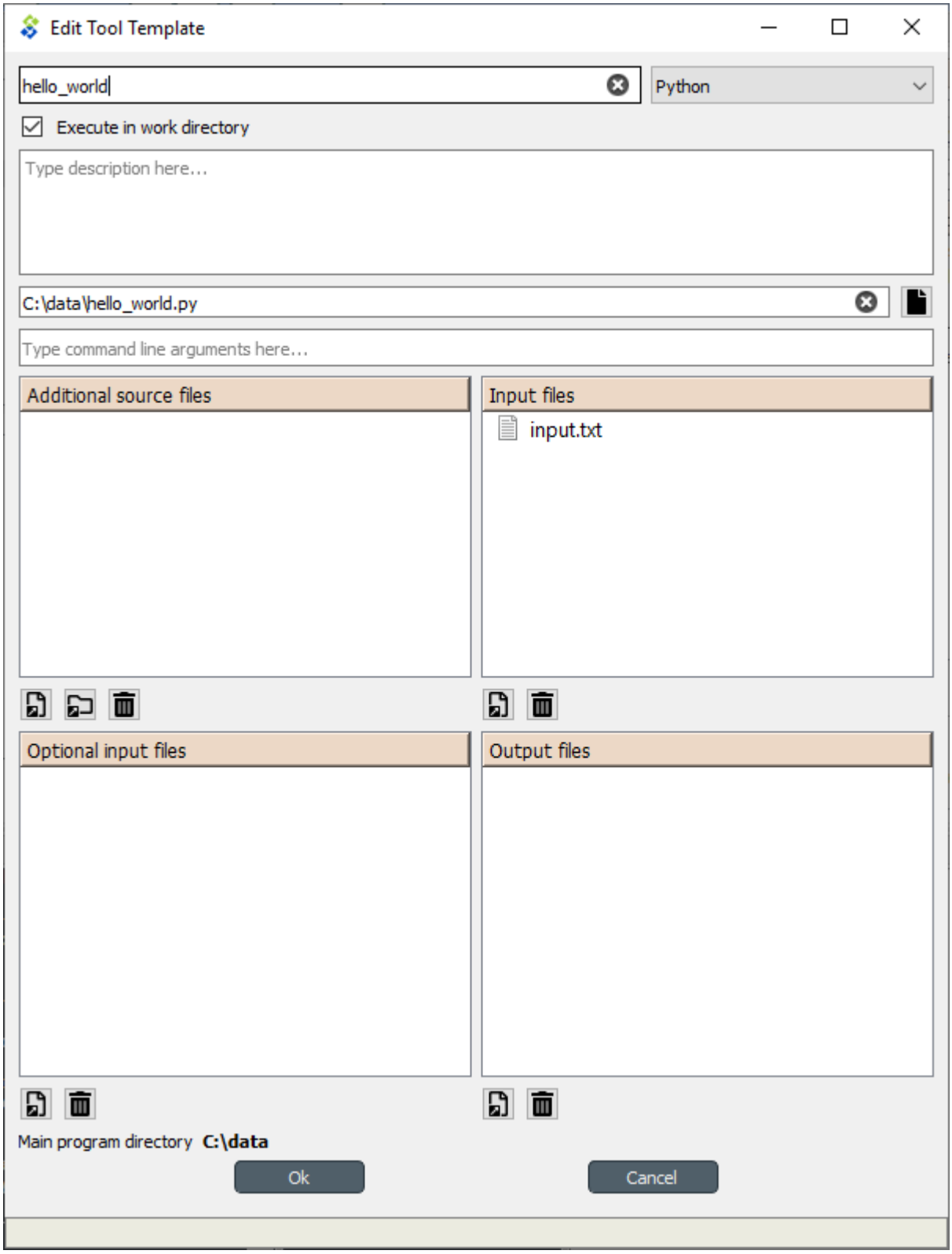
1.6 Editing a Tool template

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

Note: Input files specified in the Tool template can be used by the program source files, to obtain some relevant information for the Tool's execution. When executed, a Tool item looks for input files in **Data Connection** and **Data Store** items connected to its input.

Click on the 'tool template options' button () in 'say hello world' *Properties*, and select **Edit Tool template** from the popup menu. This will open the 'Edit Tool Template' form pre-filled with data from the 'hello_world' template.

Click the *add input files and/or directories* button right below the *Input files* list A dialog will appear that lets you can enter a name for a new input file. Type 'input.txt' and click **Ok**. The form should now be looking like this:



Click **Ok** at the bottom of the form.

Note: See *Tool template editor* for more information on editing Tool templates.

So far so good. Now let's use this input file in our program. Click on the 'tool template options' button () again, and this time select **Edit main program file...** from the popup menu. This will open the file *hello_world.py* in your default editor.

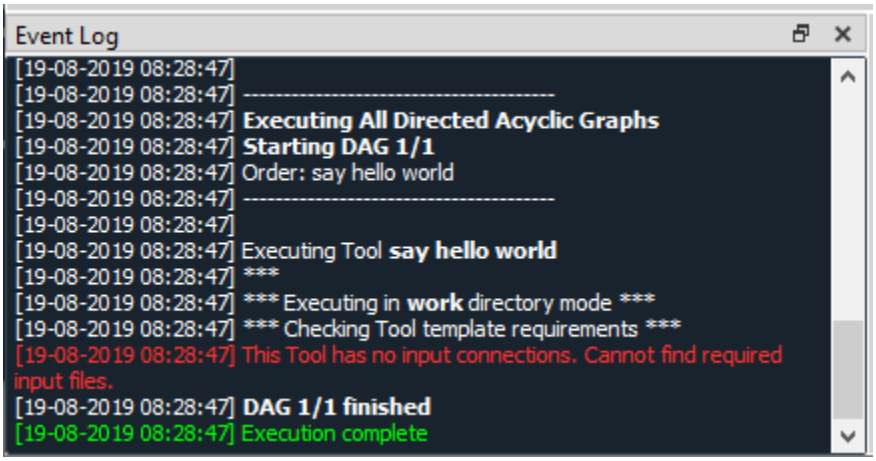
Delete whatever it's in the file and enter the following instead:

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

Save the file.

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.

Try executing the tool by pressing in the toolbar. *The execution will fail.* This is because the file 'input.txt' is not made available for the Tool:



1.7 Adding a Data Connection item to the project

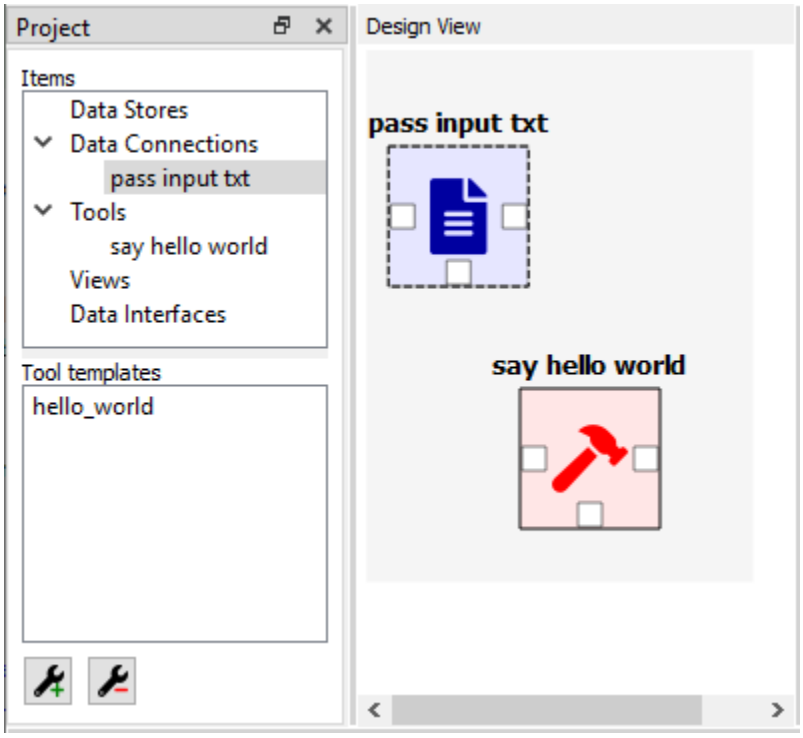
Note: The **Data Connection** item is used to hold and manipulate generic data files, so that other items, notably Tool items, can make use of that data.

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, please do one of the following:

A) From the application main menu, click **Edit -> Add Data Connection**.

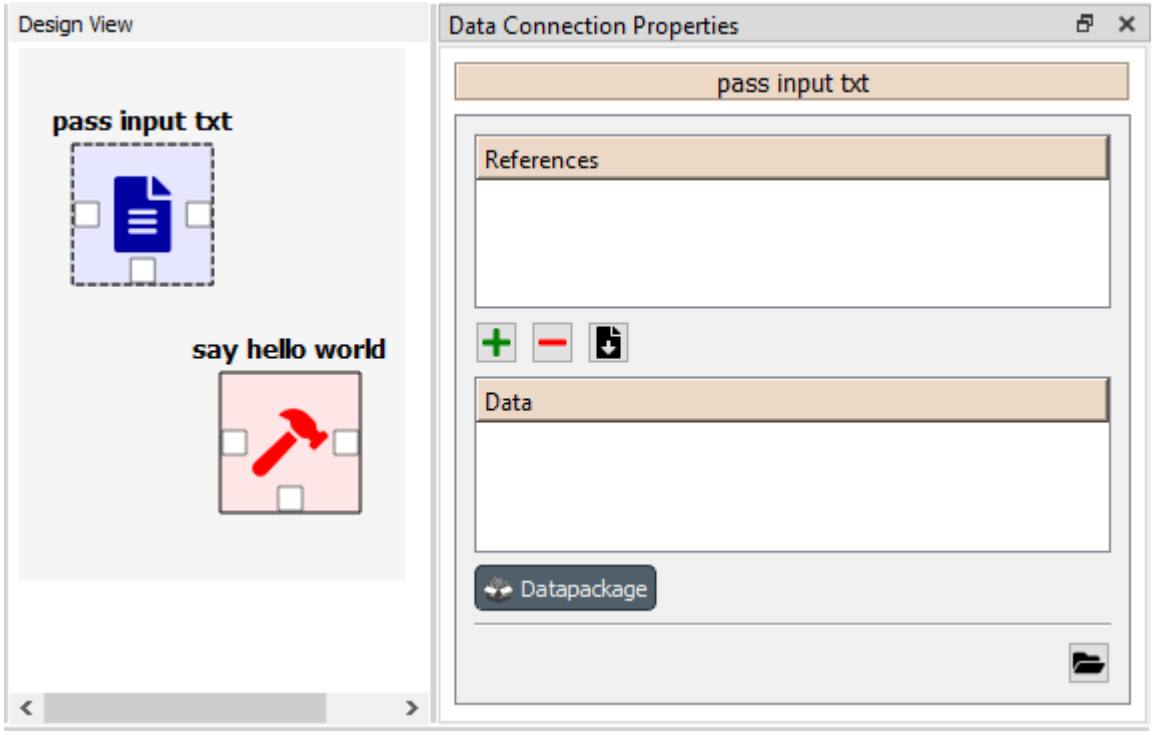
B) Drag-and-drop the Data Connection icon () from the main window toolbar onto the *Design View*.

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



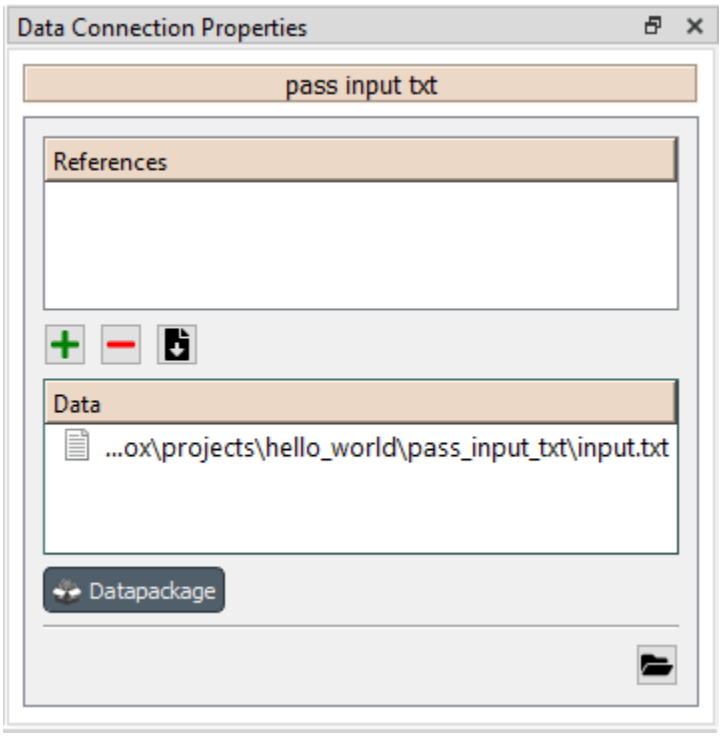
1.8 Adding data files to a Data Connection

As long as the 'pass input txt' Data Connection item is selected, you will be able to see its *Properties* on the right part of the window, looking similar to this:



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**.

Now you should see the newly created file in the *Data* list:



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

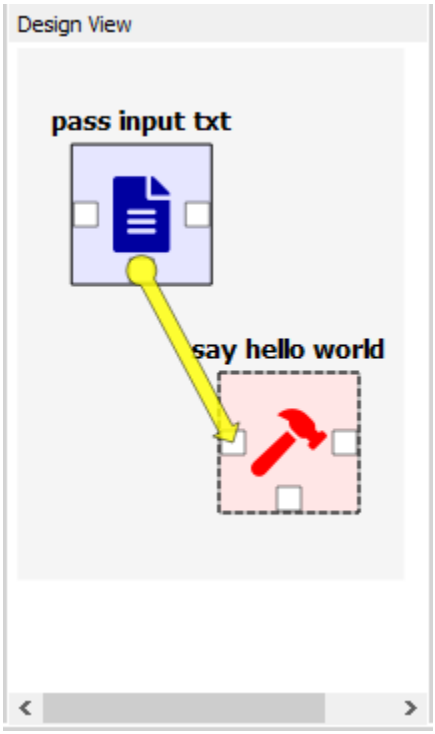
```
Hello again, World!
```

Save the file.

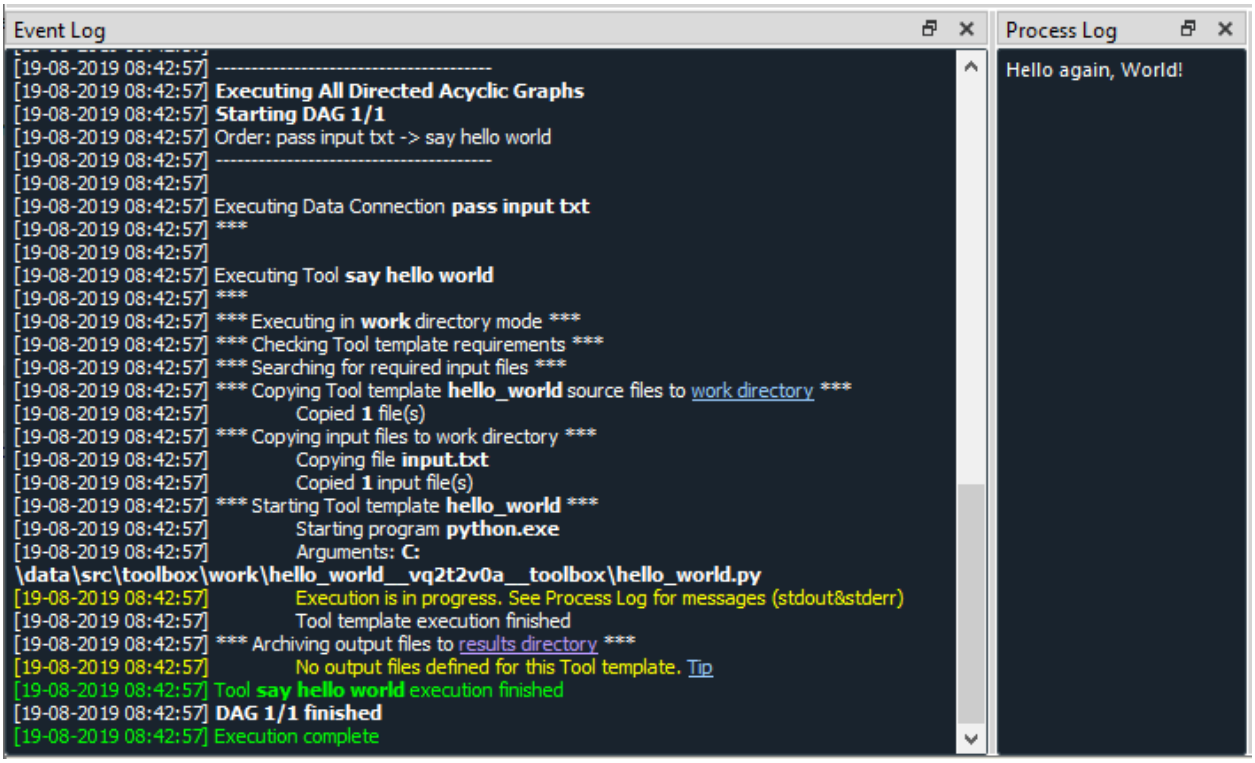
1.9 Connecting project items

As mentioned above, a Tool item looks for input files in Data Connection and Data Store items connected to its input. Thus, what we need to do now is create a *connection* from 'pass input txt' to 'say hello world', so the file 'input.txt' gets passed.

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 on the toolbar. The Tool will run 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.

CHAPTER 2

Tutorials

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

2.1 Case Study A5 tutorial

Welcome to Spine Toolbox's Case Study A5 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:

* *Configuring Julia*
* *Creating a Tool template for Spine Model*

2.1.1 Introduction

Model assumptions

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

- The capacity, or maximum electricity output. This datum also provides the maximum 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 maximum magazine level, or amount of water that can be stored in the reservoir.
- The magazine level at the beginning of the simulation period, and at the end.
- The minimum amount of water that the plant needs to discharge at every hour. This is usually zero (except for one of the plants).
- The minimum amount of water that needs to be *spilled* at every hour. Spilled water does not go through the turbine and thus does not serve to produce electricity; it just helps keeping the magazine level at bay.
- 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 total profit over the week, while respecting capacity constraints, maximum magazine level constrains, and so on. Hourly profit per plant is simply computed as the product of the electricity price and the production, minus a penalty for changes on the water discharge in two consecutive hours. This penalty is computed as the product of a constant penalty factor, common to all plants, and the absolute value of the difference in discharge with respect to the previous hour.

Modelling choices

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

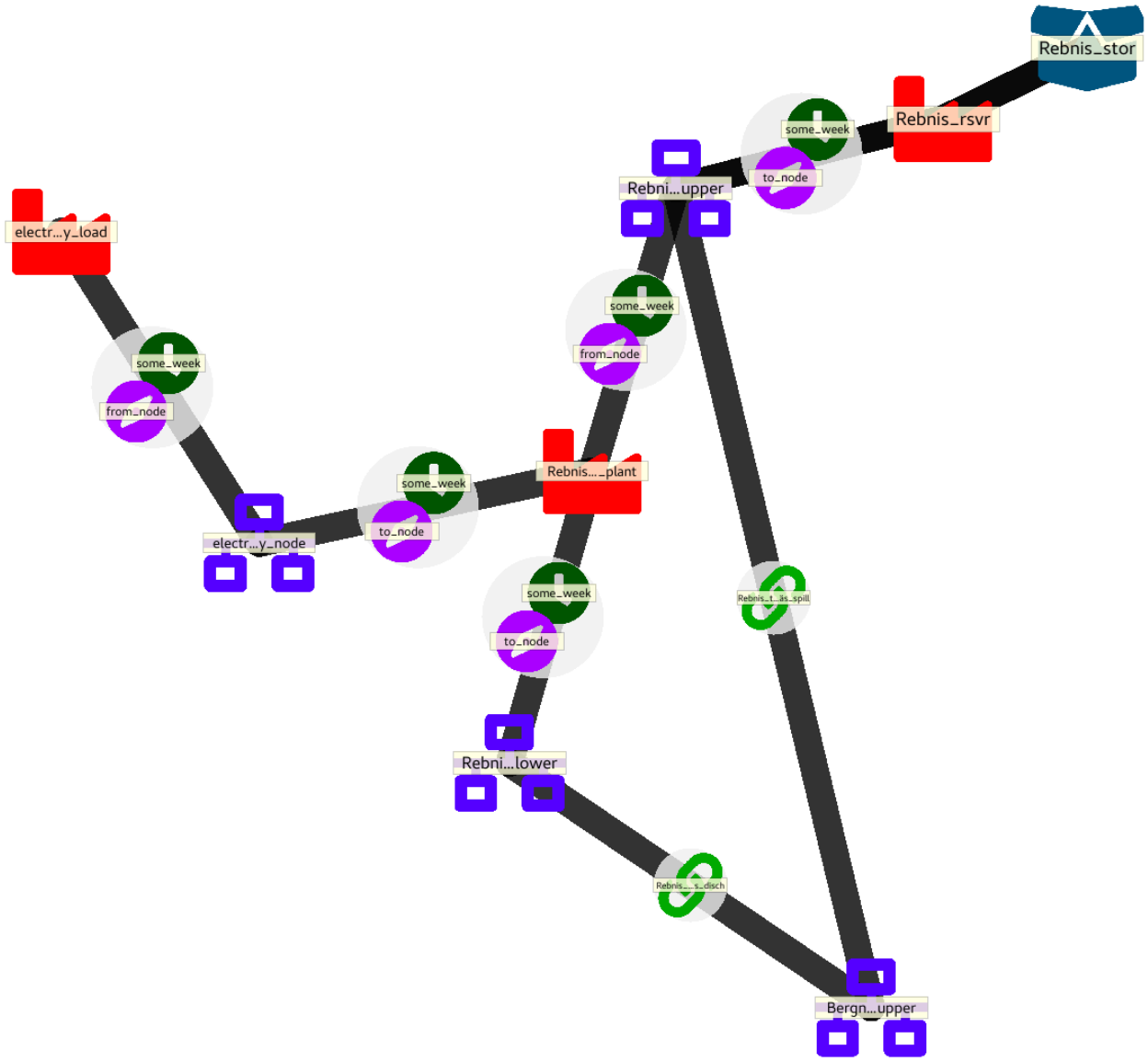
- A common electricity node.
- A load unit that takes electricity from that node.

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 reservoir unit, that takes water from the upper node to put it into a water storage and viceversa.

- 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:



2.1.2 Guide

Installing requirements

Make sure that Spine Toolbox and julia 1.0 (or greater) are properly installed as described at the following links:

- Running Spine Toolbox
- Julia downloads

Setting up project

1. Launch Spine Toolbox and from the main menu, select **File -> New...** to create a new project. Type "Case Study A5" as the project name and click **Ok**.
2. Drag the Data Store icon () from the toolbar 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 operation to create a Data Store called "output".
4. Drag the Tool icon () from the toolbar and drop it into the *Design View*. This will open the *Add Tool* dialog. Type "Spine model" as the Tool name and click **Ok**.

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

   ---

5. Click on one of "input" connectors and then on one of "Spine model" connectors. This will create a *connection* from the former to the latter.
6. Repeat the procedure to create a *connection* from "Spine model" to "output". It should look something like this:

   Todo: Add image

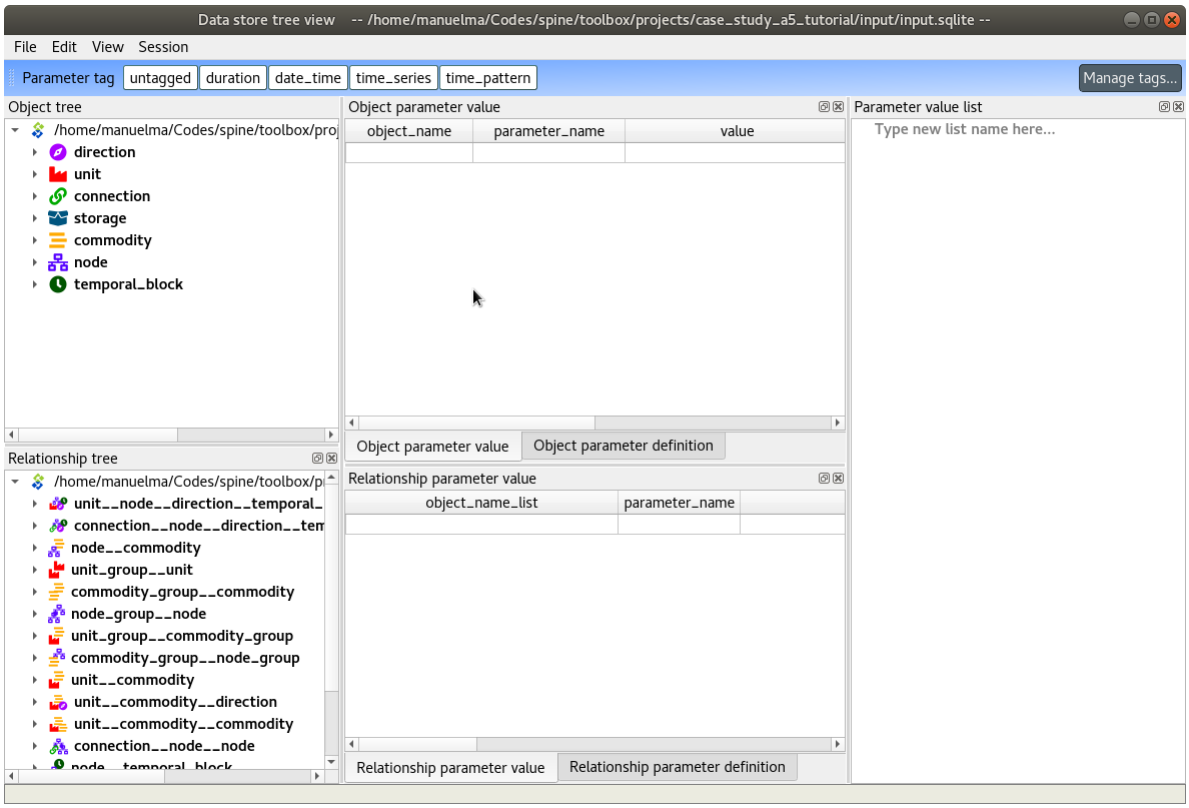
   ---

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

Entering input data

Creating input database

1. Follow the steps below to create a new Spine database for Spine Model in the 'input' Data Store:
   1. Select the 'input' Data Store item in the *Design View*.
   2. Go to *Data Store Properties*, check the box that reads **For Spine Model** and press **New Spine db**.
2. Still in *Data Store Properties*, click **Tree view**. This will open the newly created database in the *Data store tree view*, looking similar to this:



Note: The *Data store tree view* is a dedicated interface within Spine Toolbox for visualizing and managing Spine databases.

Creating objects

1. Follow the steps below to add power plants to the model as objects of class `unit`:
    1. Go to *Object tree*, right-click on `unit` and select **Add objects** from the context menu. This will open the *Add objects* dialog.
    2. With your mouse, select the list of plant names from the text-box below and copy it to the clipboard (**Ctrl+C**):

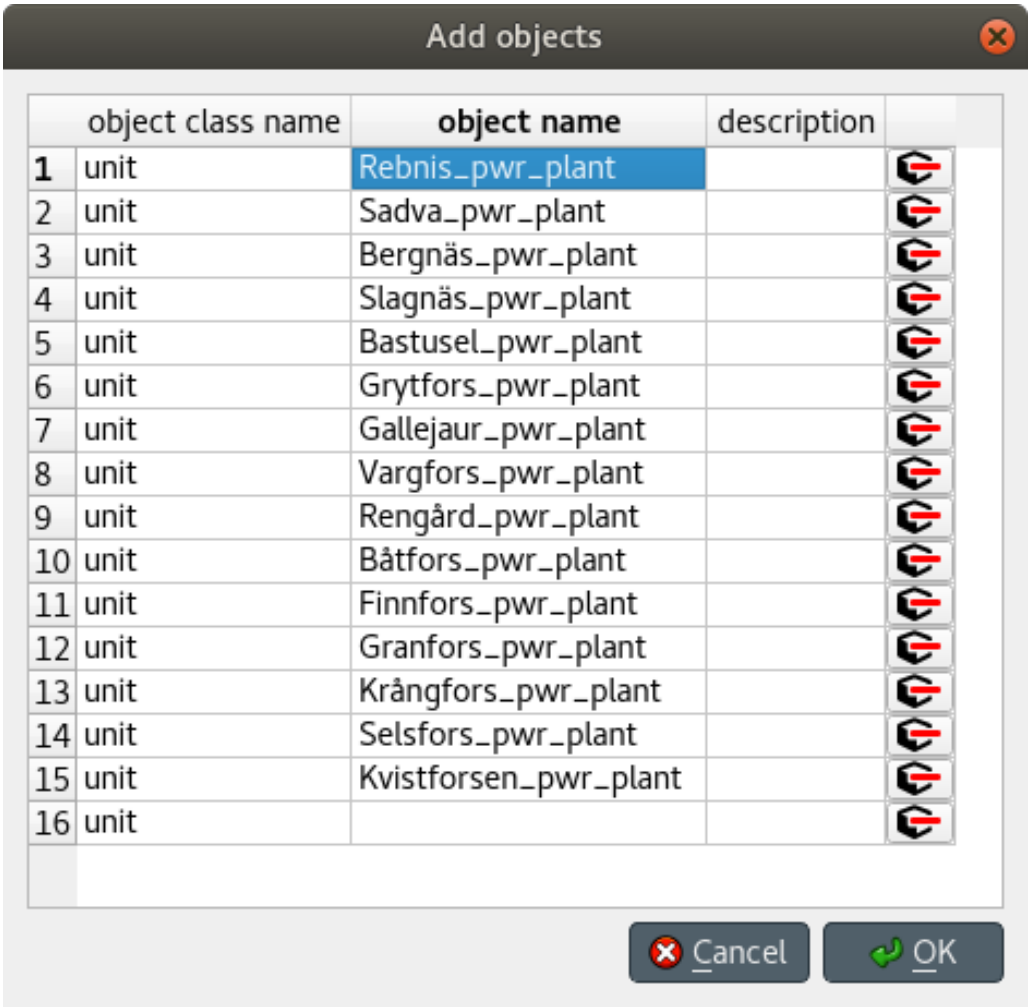
```
Rebnis_pwr_plant
Sadva_pwr_plant
Bergnäs_pwr_plant
Slagnäs_pwr_plant
Bastusel_pwr_plant
Grytfors_pwr_plant
Gallejaur_pwr_plant
Vargfors_pwr_plant
Rengård_pwr_plant
```

(continues on next page)

(continued from previous page)

```
Båtfors_pwr_plant
Finnfors_pwr_plant
Granfors_pwr_plant
Krångfors_pwr_plant
Selsfors_pwr_plant
Kvistforsen_pwr_plant
```

3. Go back to the *Add objects* dialog, 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, looking similar to this:



4. Click **Ok**.
5. Back in the *Data store tree view*, under *Object tree*, double click on `unit` to confirm that the objects are effectively there.
6. From the main menu, select **Session -> Commit** to open the *Commit changes* dialog. Enter "Add power plants" as the commit message and click **Commit**.

2. Repeat the procedure to add reservoirs as objects of class `unit`, with the following names:

```
Rebnis_rsrv
Sadva_rsrv
Bergnäs_rsrv
```

(continues on next page)

(continued from previous page)

```
Slagnäs_rsrv
Bastusel_rsrv
Grytfors_rsrv
Gallejaur_rsrv
Vargfors_rsrv
Rengård_rsrv
Båtfors_rsrv
Finnfors_rsrv
Granfors_rsrv
Krångfors_rsrv
Selsfors_rsrv
Kvistforsen_rsrv
```

3. Repeat the procedure to add discharge and spillway connections as 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
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
```

4. Repeat the procedure to add water storages as objects of class `storage`, with the following names:

```
Rebnis_stor
Sadva_stor
Bergnäs_stor
Slagnäs_stor
Bastusel_stor
Grytfors_stor
Gallejaur_stor
```

(continues on next page)

(continued from previous page)

```
Vargfors_stor
Rengård_stor
Båtfors_stor
Finnfors_stor
Granfors_stor
Krångfors_stor
Selsfors_stor
Kvistforsen_stor
```

5. Repeat the procedure to add water nodes as 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
Kvistforsen_lower
```

6. Finally, add `water` and `electricity` as objects of class `commodity`; `electricity_node` as an object of clas `node`; `electricity_load` as an object of class `unit`; and `some_week` and `past` as objects of class `temporal_block`.

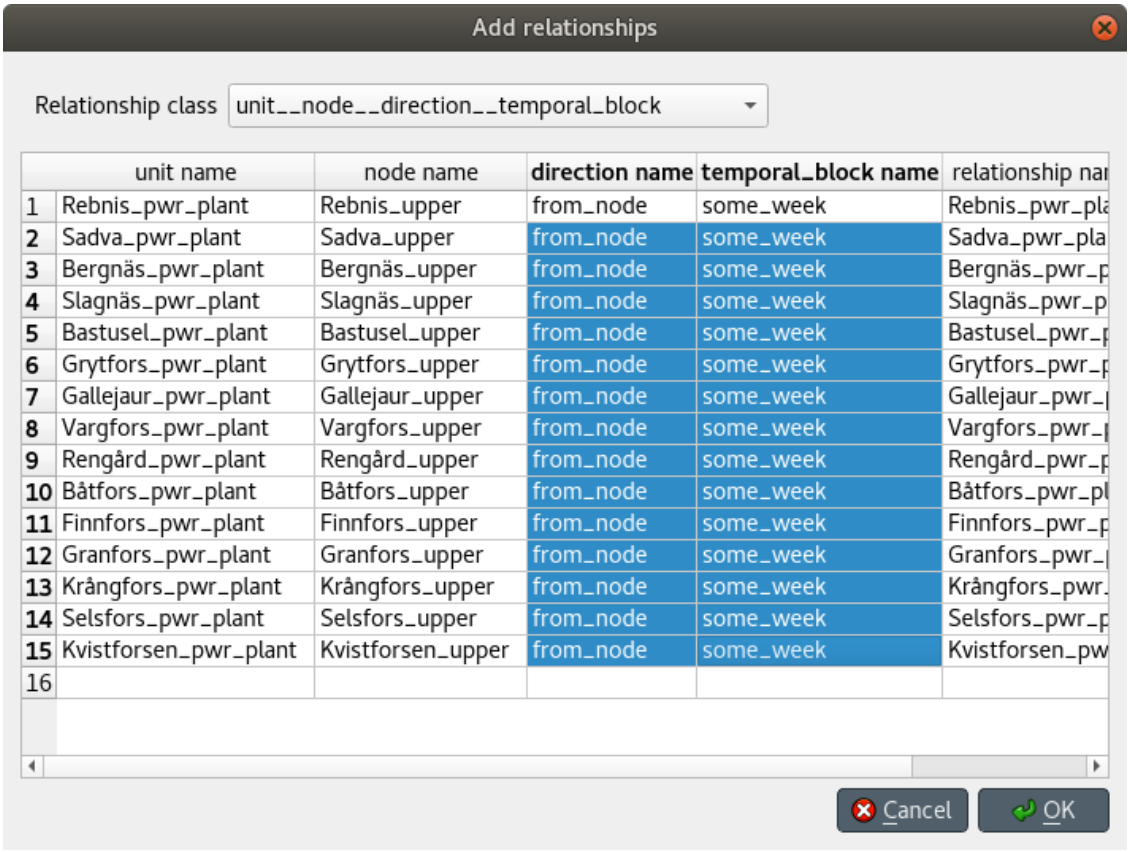
Specifying object parameter values

TODO

Establishing relationships

1. Follow the steps below to establish that power plant units receive water from the station's upper node at each time slice in the one week horizon, as relationships of class `unit__node__direction__temporal_block`:

1. Go to *Relationship tree*, right-click on `unit__node__direction__temporal_block` and select **Add relationships** from the context menu. This will open the *Add relationships* dialog.
2. Select again all *power plant names* and copy them to the clipboard (**Ctrl+C**).
3. Go back to the *Add relationships* dialog, select the first cell under the **unit name** column and press **Ctrl+V**. This will paste the list of plant names from the clipboard into that column.
4. Repeat the procedure to paste the list of *upper node names* into the **node name** column.
5. For each row in the table, enter `from_node` under **direction name** and `some_week` under **temporal block name**. Now the form should be looking like this:

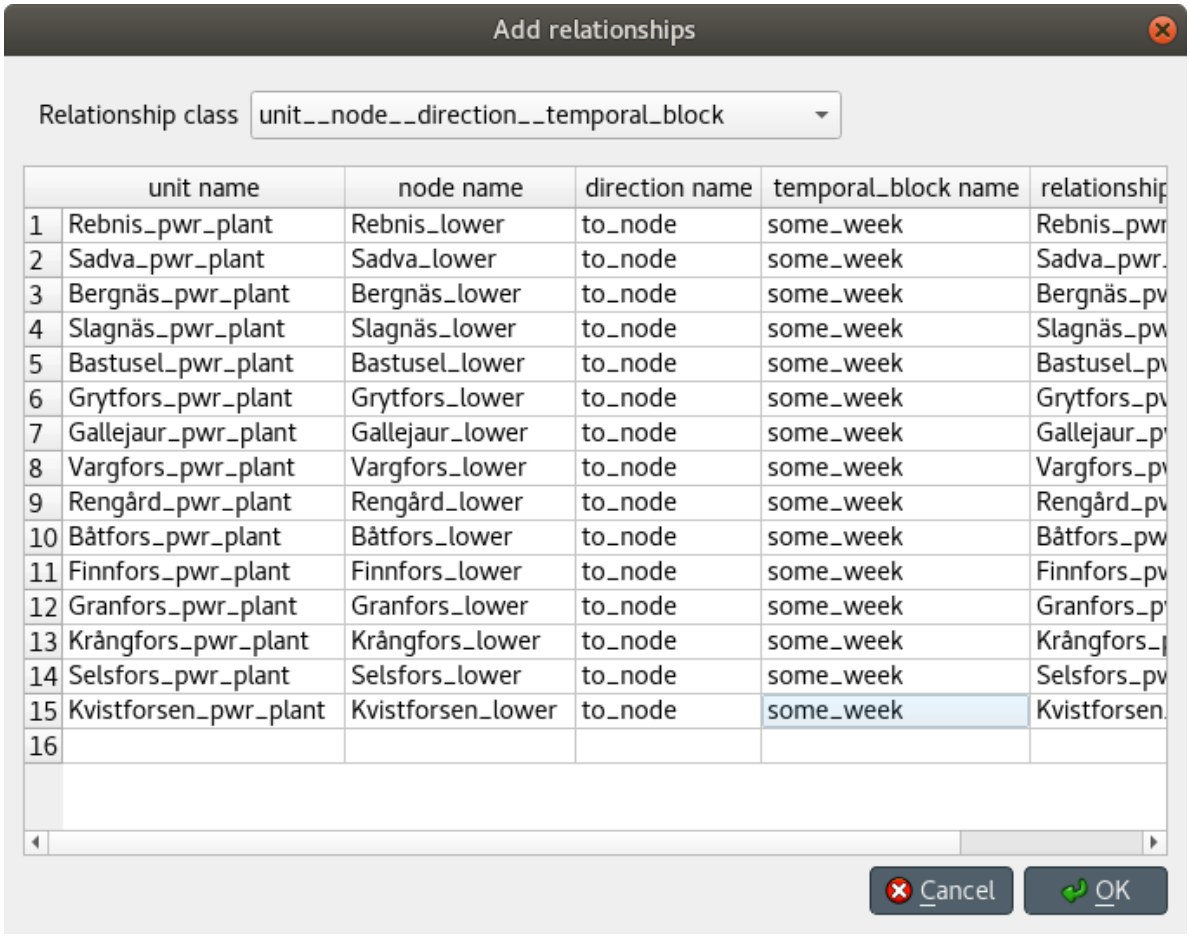


	unit name	node name	direction name	temporal_block name	relationship nar
1	Rebnis_pwr_plant	Rebnis_upper	from_node	some_week	Rebnis_pwr_pla
2	Sadva_pwr_plant	Sadva_upper	from_node	some_week	Sadva_pwr_pla
3	Bergnäs_pwr_plant	Bergnäs_upper	from_node	some_week	Bergnäs_pwr_p
4	Slagnäs_pwr_plant	Slagnäs_upper	from_node	some_week	Slagnäs_pwr_p
5	Bastusel_pwr_plant	Bastusel_upper	from_node	some_week	Bastusel_pwr_p
6	Grytfors_pwr_plant	Grytfors_upper	from_node	some_week	Grytfors_pwr_p
7	Gallejaur_pwr_plant	Gallejaur_upper	from_node	some_week	Gallejaur_pwr_p
8	Vargfors_pwr_plant	Vargfors_upper	from_node	some_week	Vargfors_pwr_p
9	Rengård_pwr_plant	Rengård_upper	from_node	some_week	Rengård_pwr_p
10	Båtfors_pwr_plant	Båtfors_upper	from_node	some_week	Båtfors_pwr_pl
11	Finnfors_pwr_plant	Finnfors_upper	from_node	some_week	Finnfors_pwr_p
12	Granfors_pwr_plant	Granfors_upper	from_node	some_week	Granfors_pwr_p
13	Krångfors_pwr_plant	Krångfors_upper	from_node	some_week	Krångfors_pwr_
14	Selsfors_pwr_plant	Selsfors_upper	from_node	some_week	Selsfors_pwr_p
15	Kvistforsen_pwr_plant	Kvistforsen_upper	from_node	some_week	Kvistforsen_pw
16					

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

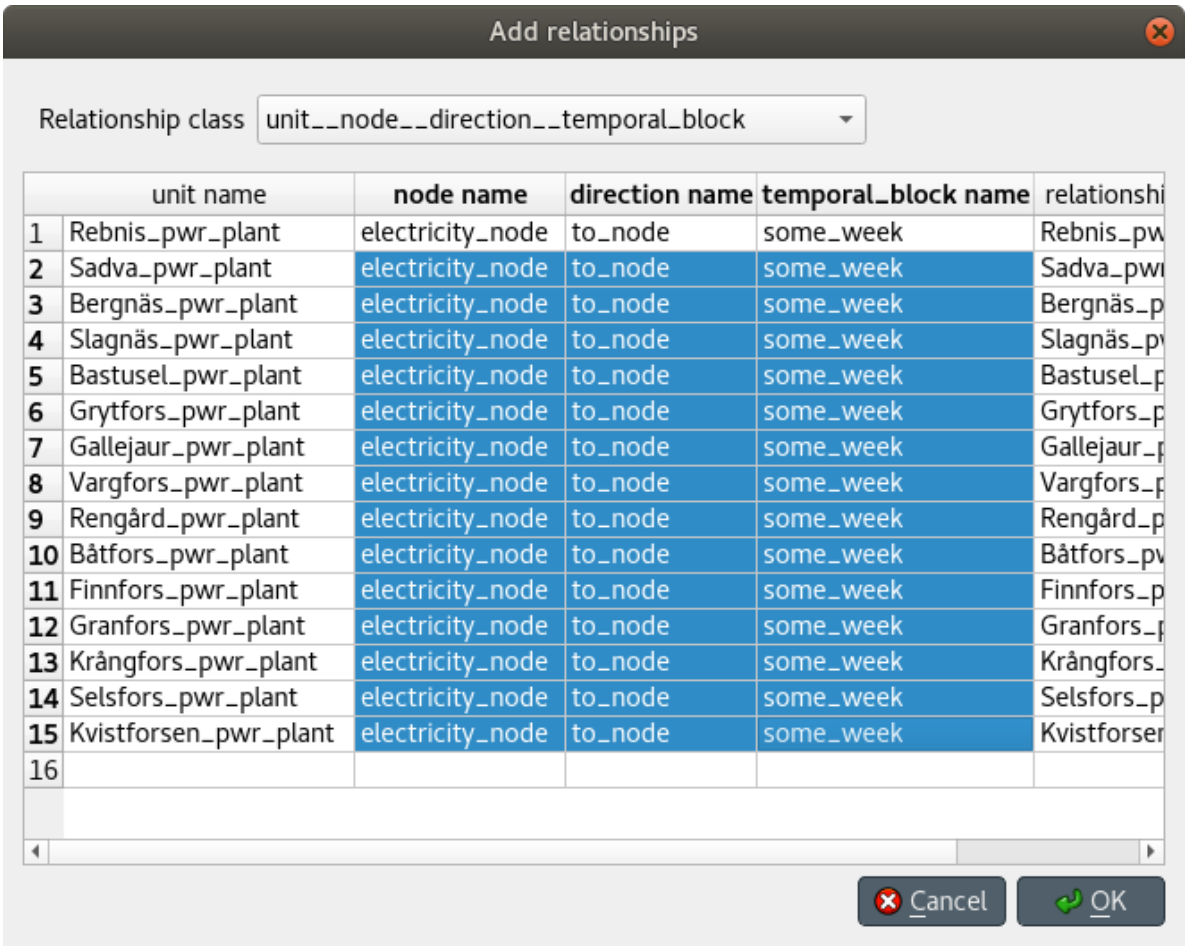
6. Click **Ok**.
7. Back in the *Data store tree view*, under *Relationship tree*, double click on `unit__node__direction__temporal_block` to confirm that the relationships are effectively there.
8. From the main menu, select **Session -> Commit** to open the *Commit changes* dialog. Enter "Add sending nodes of power plants" as the commit message and click **Commit**.

2. Repeat the procedure to establish that power plant units release water to the station's lower node at each time slice in the one week horizon, as relationships of class `unit__node__direction__temporal_block`:

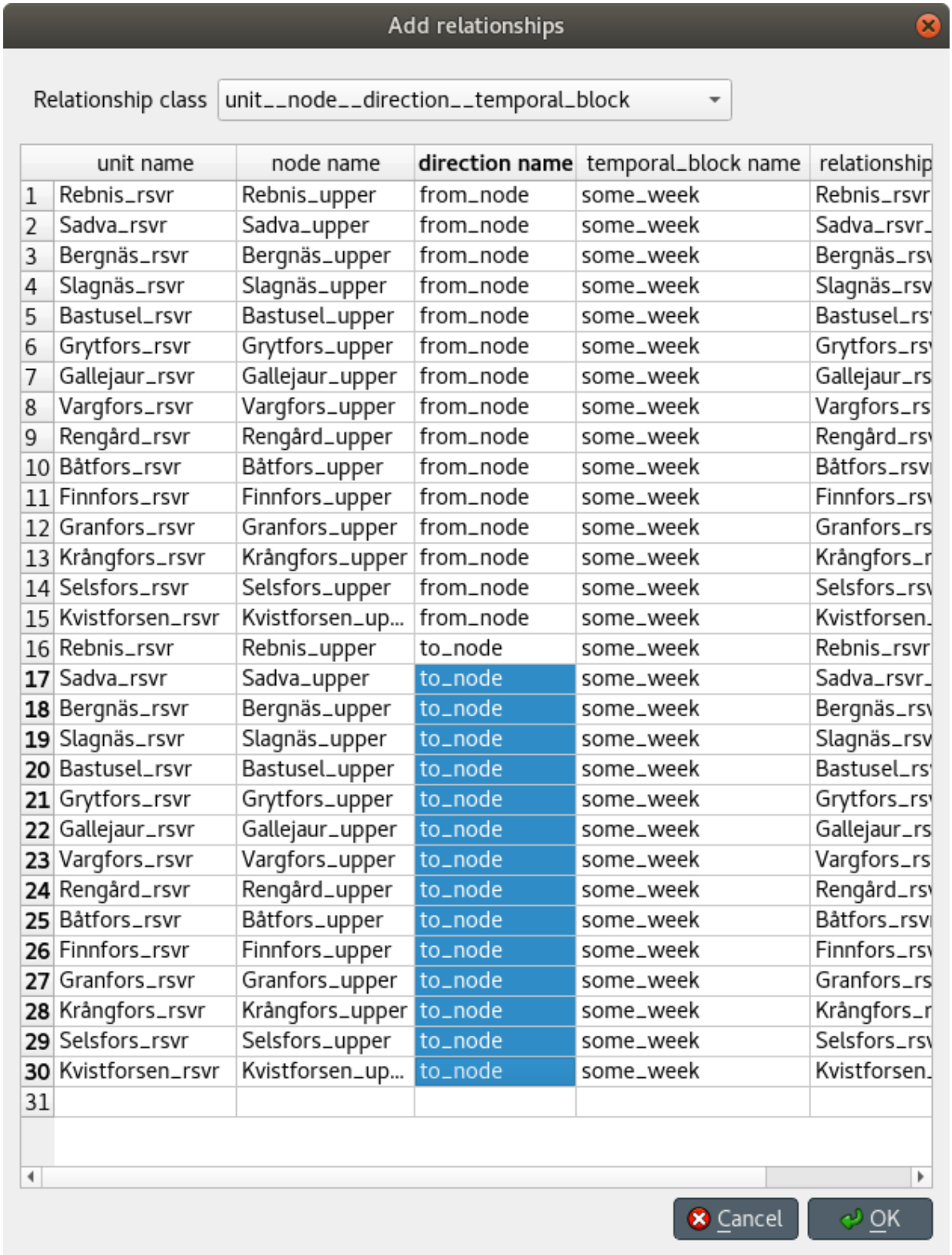


	unit name	node name	direction name	temporal_block name	relationship
1	Rebnis_pwr_plant	Rebnis_lower	to_node	some_week	Rebnis_pwr
2	Sadva_pwr_plant	Sadva_lower	to_node	some_week	Sadva_pwr
3	Bergnäs_pwr_plant	Bergnäs_lower	to_node	some_week	Bergnäs_pv
4	Slagnäs_pwr_plant	Slagnäs_lower	to_node	some_week	Slagnäs_pw
5	Bastusel_pwr_plant	Bastusel_lower	to_node	some_week	Bastusel_pv
6	Grytfors_pwr_plant	Grytfors_lower	to_node	some_week	Grytfors_pv
7	Gallejaur_pwr_plant	Gallejaur_lower	to_node	some_week	Gallejaur_p
8	Vargfors_pwr_plant	Vargfors_lower	to_node	some_week	Vargfors_pv
9	Rengård_pwr_plant	Rengård_lower	to_node	some_week	Rengård_pv
10	Båtfors_pwr_plant	Båtfors_lower	to_node	some_week	Båtfors_pw
11	Finnfors_pwr_plant	Finnfors_lower	to_node	some_week	Finnfors_pv
12	Granfors_pwr_plant	Granfors_lower	to_node	some_week	Granfors_p
13	Krångfors_pwr_plant	Krångfors_lower	to_node	some_week	Krångfors_
14	Selsfors_pwr_plant	Selsfors_lower	to_node	some_week	Selsfors_pv
15	Kvistforsen_pwr_plant	Kvistforsen_lower	to_node	some_week	Kvistforsen
16					

3. Repeat the procedure to establish that power plant units release electricity to the common electricity node at each time slice in the one week horizon, as relationships of class `unit__node__direction__temporal_block`:

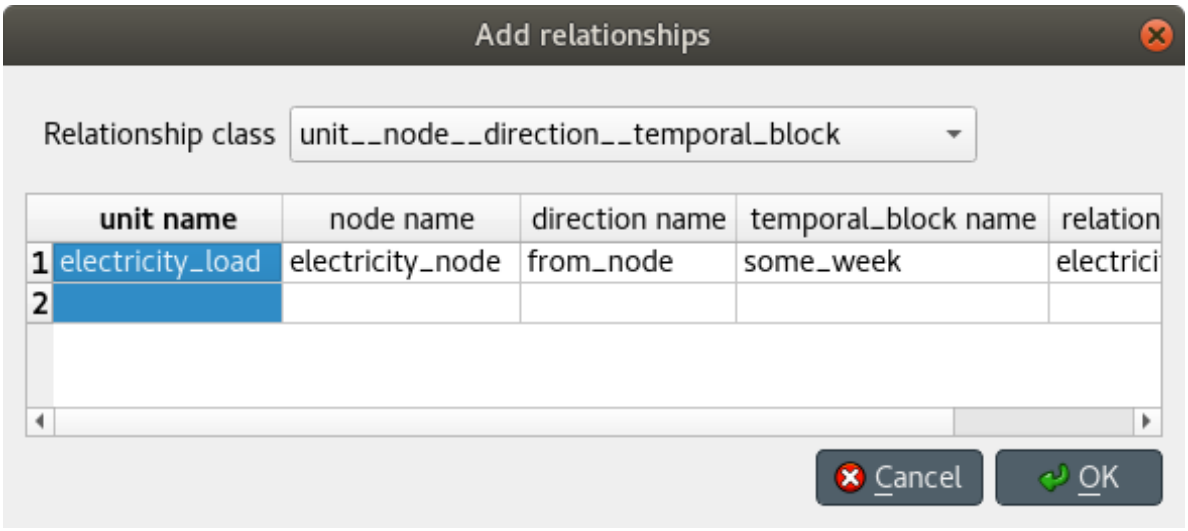


4. Repeat the procedure to establish that reservoir units take and release water to and from the station's upper node at each time slice in the one week horizon, as relationships of class `unit__node__direction__temporal_block`:

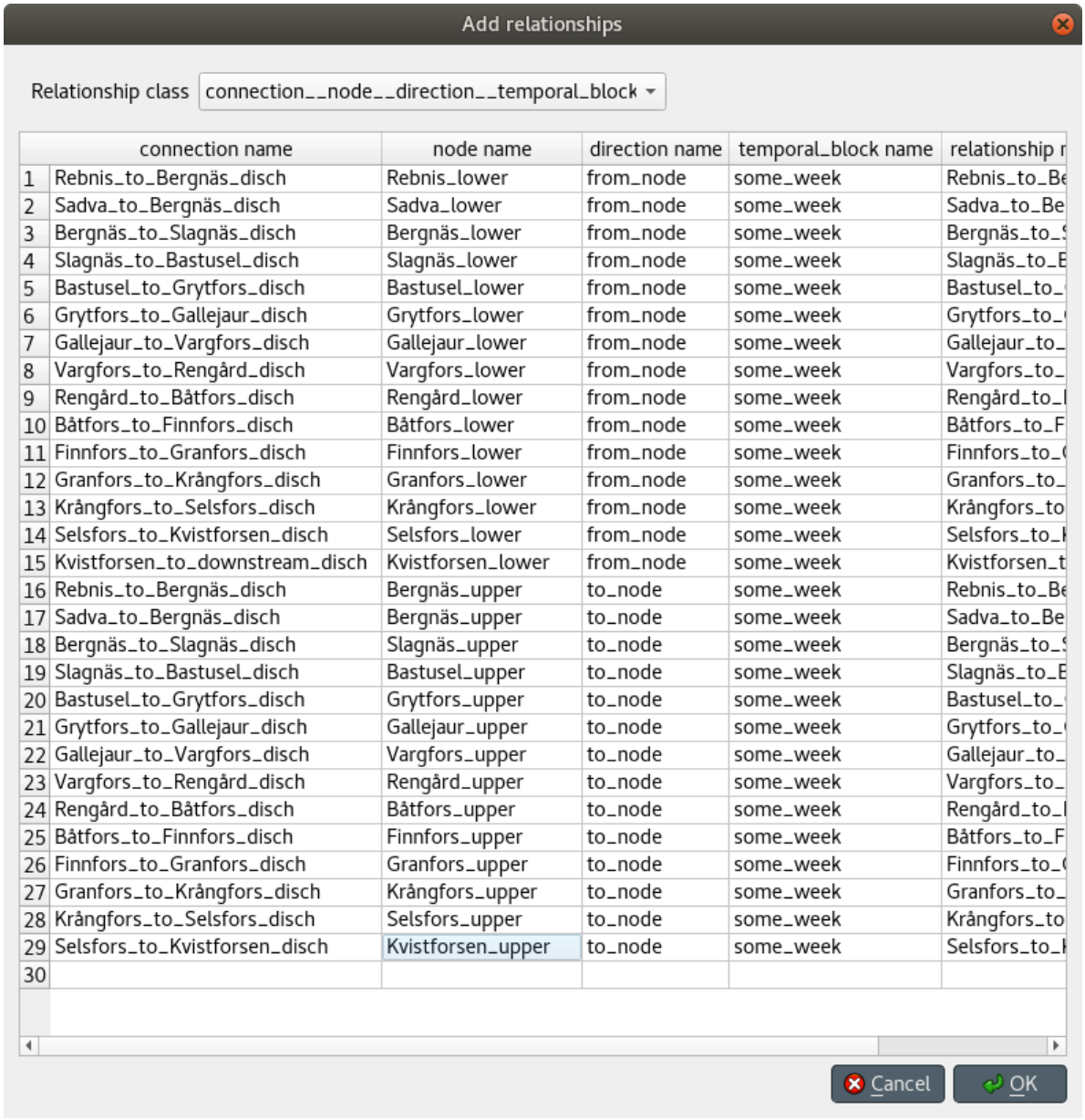


	unit name	node name	direction name	temporal_block name	relationship
1	Rebnis_rsvr	Rebnis_upper	from_node	some_week	Rebnis_rsvr
2	Sadva_rsvr	Sadva_upper	from_node	some_week	Sadva_rsvr_
3	Bergnäs_rsvr	Bergnäs_upper	from_node	some_week	Bergnäs_rsv
4	Slagnäs_rsvr	Slagnäs_upper	from_node	some_week	Slagnäs_rsv
5	Bastusel_rsvr	Bastusel_upper	from_node	some_week	Bastusel_rs
6	Grytfors_rsvr	Grytfors_upper	from_node	some_week	Grytfors_rs
7	Gallejaur_rsvr	Gallejaur_upper	from_node	some_week	Gallejaur_rs
8	Vargfors_rsvr	Vargfors_upper	from_node	some_week	Vargfors_rs
9	Rengård_rsvr	Rengård_upper	from_node	some_week	Rengård_rsv
10	Båtfors_rsvr	Båtfors_upper	from_node	some_week	Båtfors_rsv
11	Finnfors_rsvr	Finnfors_upper	from_node	some_week	Finnfors_rsv
12	Granfors_rsvr	Granfors_upper	from_node	some_week	Granfors_rs
13	Krångfors_rsvr	Krångfors_upper	from_node	some_week	Krångfors_r
14	Selsfors_rsvr	Selsfors_upper	from_node	some_week	Selsfors_rsv
15	Kvistforsen_rsvr	Kvistforsen_up...	from_node	some_week	Kvistforsen_
16	Rebnis_rsvr	Rebnis_upper	to_node	some_week	Rebnis_rsvr
17	Sadva_rsvr	Sadva_upper	to_node	some_week	Sadva_rsvr_
18	Bergnäs_rsvr	Bergnäs_upper	to_node	some_week	Bergnäs_rsv
19	Slagnäs_rsvr	Slagnäs_upper	to_node	some_week	Slagnäs_rsv
20	Bastusel_rsvr	Bastusel_upper	to_node	some_week	Bastusel_rs
21	Grytfors_rsvr	Grytfors_upper	to_node	some_week	Grytfors_rs
22	Gallejaur_rsvr	Gallejaur_upper	to_node	some_week	Gallejaur_rs
23	Vargfors_rsvr	Vargfors_upper	to_node	some_week	Vargfors_rs
24	Rengård_rsvr	Rengård_upper	to_node	some_week	Rengård_rsv
25	Båtfors_rsvr	Båtfors_upper	to_node	some_week	Båtfors_rsv
26	Finnfors_rsvr	Finnfors_upper	to_node	some_week	Finnfors_rsv
27	Granfors_rsvr	Granfors_upper	to_node	some_week	Granfors_rs
28	Krångfors_rsvr	Krångfors_upper	to_node	some_week	Krångfors_r
29	Selsfors_rsvr	Selsfors_upper	to_node	some_week	Selsfors_rsv
30	Kvistforsen_rsvr	Kvistforsen_up...	to_node	some_week	Kvistforsen_
31					

5. Repeat the procedure to establish that the electricity load takes electricity from the common electricity node at each time slice in the one week horizon, as a relationship of class `unit__node__direction__temporal_block`:

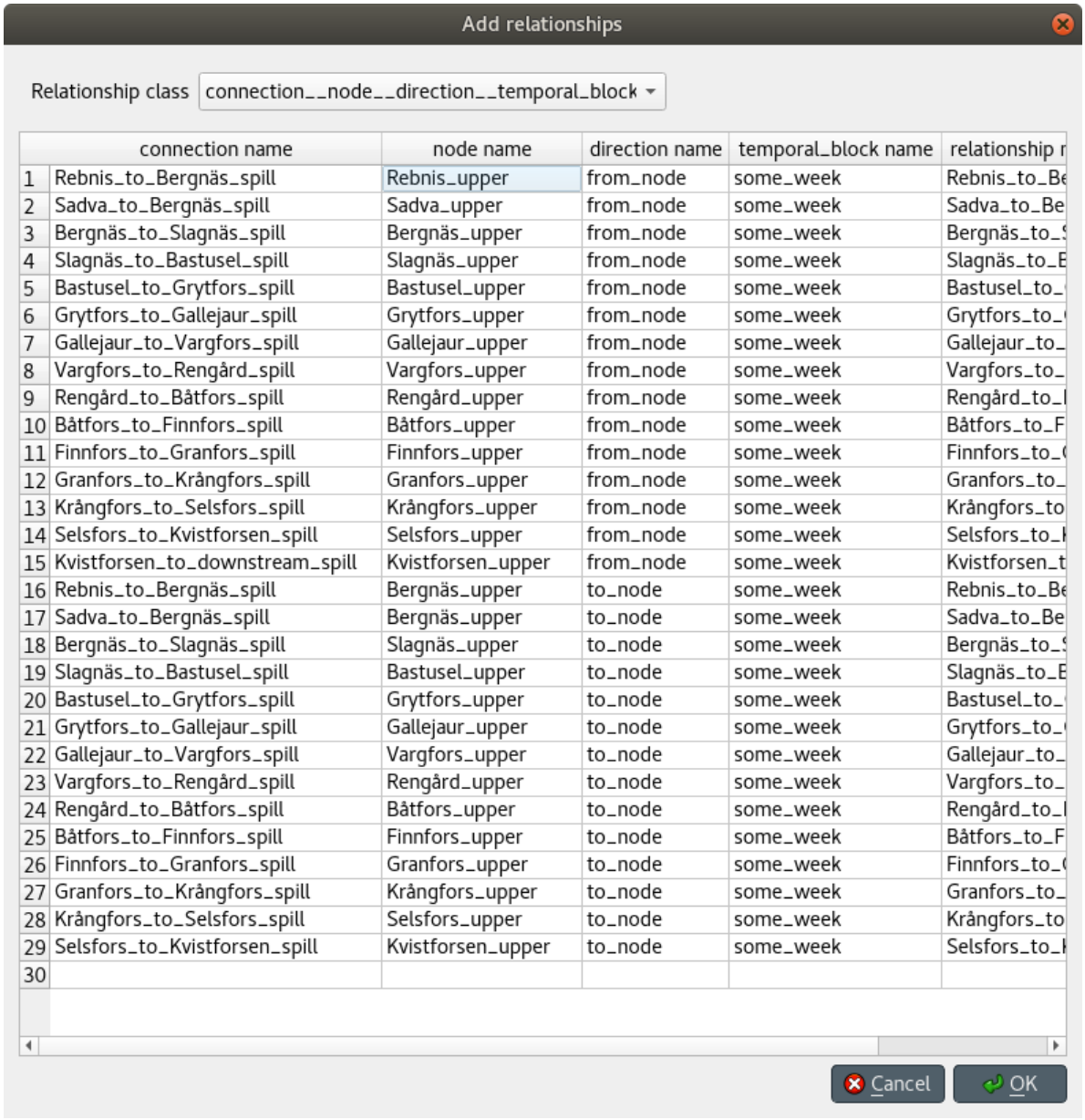


6. Repeat the procedure to establish that discharge connections take water from the lower node of one station and release it to the upper node of the downstream station, at each time slice in the one week horizon, as relationships of class `connection__node__direction__temporal_block`:



	connection name	node name	direction name	temporal_block name	relationship r
1	Rebnis_to_Bergnäs_disch	Rebnis_lower	from_node	some_week	Rebnis_to_Be
2	Sadva_to_Bergnäs_disch	Sadva_lower	from_node	some_week	Sadva_to_Be
3	Bergnäs_to_Slagnäs_disch	Bergnäs_lower	from_node	some_week	Bergnäs_to_S
4	Slagnäs_to_Bastusel_disch	Slagnäs_lower	from_node	some_week	Slagnäs_to_B
5	Bastusel_to_Grytfors_disch	Bastusel_lower	from_node	some_week	Bastusel_to_
6	Grytfors_to_Gallejaur_disch	Grytfors_lower	from_node	some_week	Grytfors_to_
7	Gallejaur_to_Vargfors_disch	Gallejaur_lower	from_node	some_week	Gallejaur_to_
8	Vargfors_to_Rengård_disch	Vargfors_lower	from_node	some_week	Vargfors_to_
9	Rengård_to_Båtfors_disch	Rengård_lower	from_node	some_week	Rengård_to_B
10	Båtfors_to_Finnfors_disch	Båtfors_lower	from_node	some_week	Båtfors_to_F
11	Finnfors_to_Granfors_disch	Finnfors_lower	from_node	some_week	Finnfors_to_G
12	Granfors_to_Krångfors_disch	Granfors_lower	from_node	some_week	Granfors_to_
13	Krångfors_to_Selsfors_disch	Krångfors_lower	from_node	some_week	Krångfors_to
14	Selsfors_to_Kvistforsen_disch	Selsfors_lower	from_node	some_week	Selsfors_to_K
15	Kvistforsen_to_downstream_disch	Kvistforsen_lower	from_node	some_week	Kvistforsen_t
16	Rebnis_to_Bergnäs_disch	Bergnäs_upper	to_node	some_week	Rebnis_to_Be
17	Sadva_to_Bergnäs_disch	Bergnäs_upper	to_node	some_week	Sadva_to_Be
18	Bergnäs_to_Slagnäs_disch	Slagnäs_upper	to_node	some_week	Bergnäs_to_S
19	Slagnäs_to_Bastusel_disch	Bastusel_upper	to_node	some_week	Slagnäs_to_B
20	Bastusel_to_Grytfors_disch	Grytfors_upper	to_node	some_week	Bastusel_to_
21	Grytfors_to_Gallejaur_disch	Gallejaur_upper	to_node	some_week	Grytfors_to_G
22	Gallejaur_to_Vargfors_disch	Vargfors_upper	to_node	some_week	Gallejaur_to_
23	Vargfors_to_Rengård_disch	Rengård_upper	to_node	some_week	Vargfors_to_
24	Rengård_to_Båtfors_disch	Båtfors_upper	to_node	some_week	Rengård_to_B
25	Båtfors_to_Finnfors_disch	Finnfors_upper	to_node	some_week	Båtfors_to_F
26	Finnfors_to_Granfors_disch	Granfors_upper	to_node	some_week	Finnfors_to_G
27	Granfors_to_Krångfors_disch	Krångfors_upper	to_node	some_week	Granfors_to_
28	Krångfors_to_Selsfors_disch	Selsfors_upper	to_node	some_week	Krångfors_to
29	Selsfors_to_Kvistforsen_disch	Kvistforsen_upper	to_node	some_week	Selsfors_to_K
30					

7. Repeat the procedure to establish that spillway connections take water from the upper node of one station and release it to the upper node of the downstream station, at each time slice in the one week horizon, as relationships of class `connection__node__direction__temporal_block`:



	connection name	node name	direction name	temporal_block name	relationship r
1	Rebnis_to_Bergnäs_spill	Rebnis_upper	from_node	some_week	Rebnis_to_Be
2	Sadva_to_Bergnäs_spill	Sadva_upper	from_node	some_week	Sadva_to_Be
3	Bergnäs_to_Slagnäs_spill	Bergnäs_upper	from_node	some_week	Bergnäs_to_S
4	Slagnäs_to_Bastusel_spill	Slagnäs_upper	from_node	some_week	Slagnäs_to_B
5	Bastusel_to_Grytfors_spill	Bastusel_upper	from_node	some_week	Bastusel_to_
6	Grytfors_to_Gallejaur_spill	Grytfors_upper	from_node	some_week	Grytfors_to_(
7	Gallejaur_to_Vargfors_spill	Gallejaur_upper	from_node	some_week	Gallejaur_to_
8	Vargfors_to_Rengård_spill	Vargfors_upper	from_node	some_week	Vargfors_to_
9	Rengård_to_Båtfors_spill	Rengård_upper	from_node	some_week	Rengård_to_I
10	Båtfors_to_Finnfors_spill	Båtfors_upper	from_node	some_week	Båtfors_to_F
11	Finnfors_to_Granfors_spill	Finnfors_upper	from_node	some_week	Finnfors_to_(
12	Granfors_to_Krångfors_spill	Granfors_upper	from_node	some_week	Granfors_to_
13	Krångfors_to_Selsfors_spill	Krångfors_upper	from_node	some_week	Krångfors_to
14	Selsfors_to_Kvistforsen_spill	Selsfors_upper	from_node	some_week	Selsfors_to_K
15	Kvistforsen_to_downstream_spill	Kvistforsen_upper	from_node	some_week	Kvistforsen_t
16	Rebnis_to_Bergnäs_spill	Bergnäs_upper	to_node	some_week	Rebnis_to_Be
17	Sadva_to_Bergnäs_spill	Bergnäs_upper	to_node	some_week	Sadva_to_Be
18	Bergnäs_to_Slagnäs_spill	Slagnäs_upper	to_node	some_week	Bergnäs_to_S
19	Slagnäs_to_Bastusel_spill	Bastusel_upper	to_node	some_week	Slagnäs_to_B
20	Bastusel_to_Grytfors_spill	Grytfors_upper	to_node	some_week	Bastusel_to_
21	Grytfors_to_Gallejaur_spill	Gallejaur_upper	to_node	some_week	Grytfors_to_(
22	Gallejaur_to_Vargfors_spill	Vargfors_upper	to_node	some_week	Gallejaur_to_
23	Vargfors_to_Rengård_spill	Rengård_upper	to_node	some_week	Vargfors_to_
24	Rengård_to_Båtfors_spill	Båtfors_upper	to_node	some_week	Rengård_to_I
25	Båtfors_to_Finnfors_spill	Finnfors_upper	to_node	some_week	Båtfors_to_F
26	Finnfors_to_Granfors_spill	Granfors_upper	to_node	some_week	Finnfors_to_(
27	Granfors_to_Krångfors_spill	Krångfors_upper	to_node	some_week	Granfors_to_
28	Krångfors_to_Selsfors_spill	Selsfors_upper	to_node	some_week	Krångfors_to
29	Selsfors_to_Kvistforsen_spill	Kvistforsen_upper	to_node	some_week	Selsfors_to_K
30					

8. Repeat the procedure to establish that water nodes balance water, and the electricity node balances electricity, as relationships of class `node__commodity`:

Add relationships

Relationship class: node__commodity

	node name	commodity name	relationship name
1	Rebnis_upper	water	Rebnis_upper_...
2	Sadva_upper	water	Sadva_upper_...
3	Bergnäs_upper	water	Bergnäs_upper_...
4	Slagnäs_upper	water	Slagnäs_upper_...
5	Bastusel_upper	water	Bastusel_upper...
6	Grytfors_upper	water	Grytfors_upper_...
7	Gallejaur_upper	water	Gallejaur_upper...
8	Vargfors_upper	water	Vargfors_upper...
9	Rengård_upper	water	Rengård_upper_...
10	Båtfors_upper	water	Båtfors_upper_...
11	Finnfors_upper	water	Finnfors_upper_...
12	Granfors_upper	water	Granfors_upper...
13	Krångfors_upper	water	Krångfors_uppe...
14	Selsfors_upper	water	Selsfors_upper_...
15	Kvistforsen_upper	water	Kvistforsen_upp...
16	Rebnis_lower	water	Rebnis_lower_...
17	Sadva_lower	water	Sadva_lower_...
18	Bergnäs_lower	water	Bergnäs_lower_...
19	Slagnäs_lower	water	Slagnäs_lower_...
20	Bastusel_lower	water	Bastusel_lower_...
21	Grytfors_lower	water	Grytfors_lower_...
22	Gallejaur_lower	water	Gallejaur_lower...
23	Vargfors_lower	water	Vargfors_lower...
24	Rengård_lower	water	Rengård_lower_...
25	Båtfors_lower	water	Båtfors_lower_...
26	Finnfors_lower	water	Finnfors_lower_...
27	Granfors_lower	water	Granfors_lower...
28	Krångfors_lower	water	Krångfors_lowe...
29	Selsfors_lower	water	Selsfors_lower_...
30	Kvistforsen_lower	water	Kvistforsen_low...
31	electricity_node	electricity	electricity_node...
32			

Cancel OK

9. Repeat the procedure to establish that all nodes are balanced at each time slice in the one week horizon, as relationships of class `node__temporal_block`:

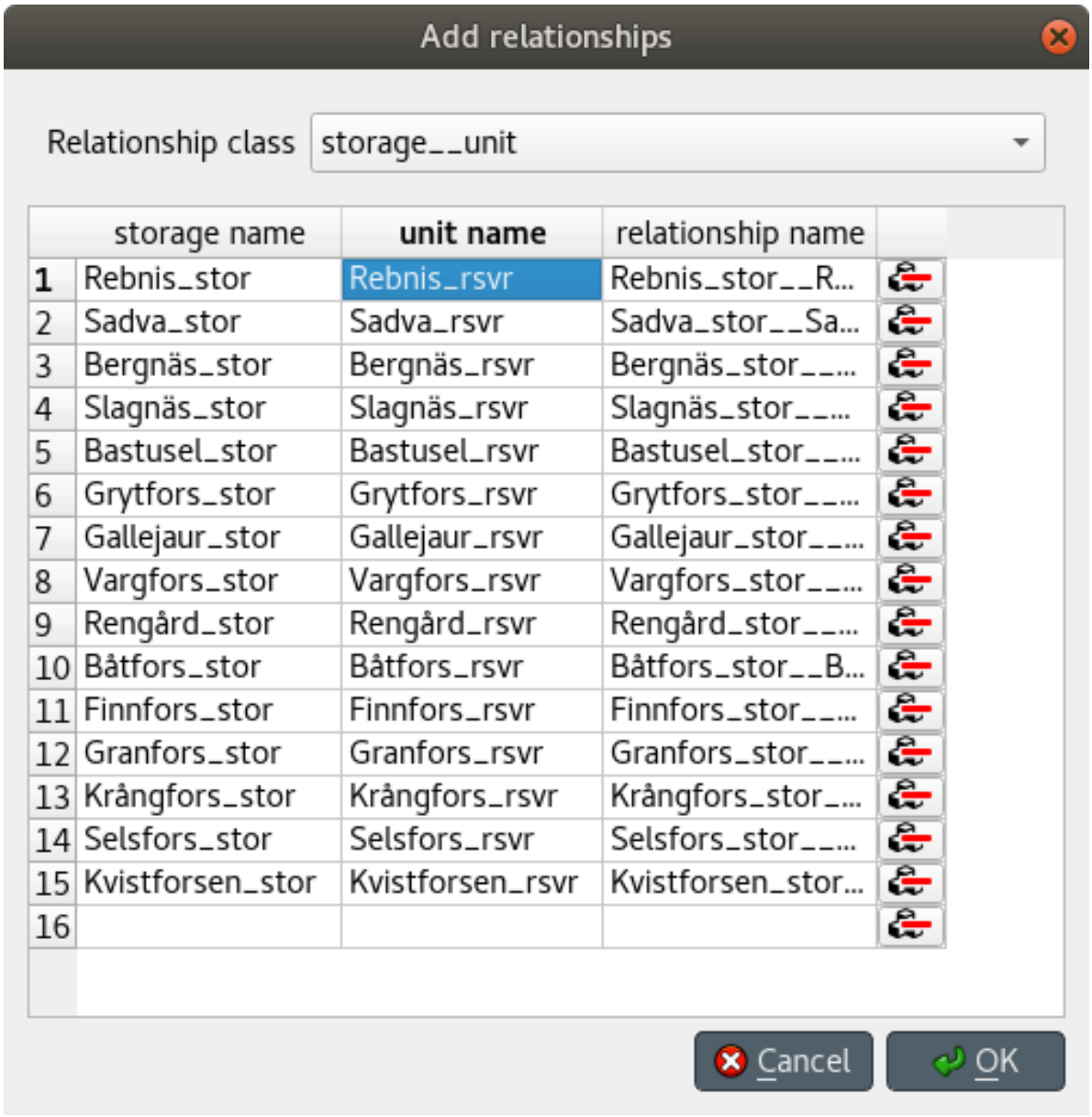
Add relationships

Relationship class: node__temporal_block

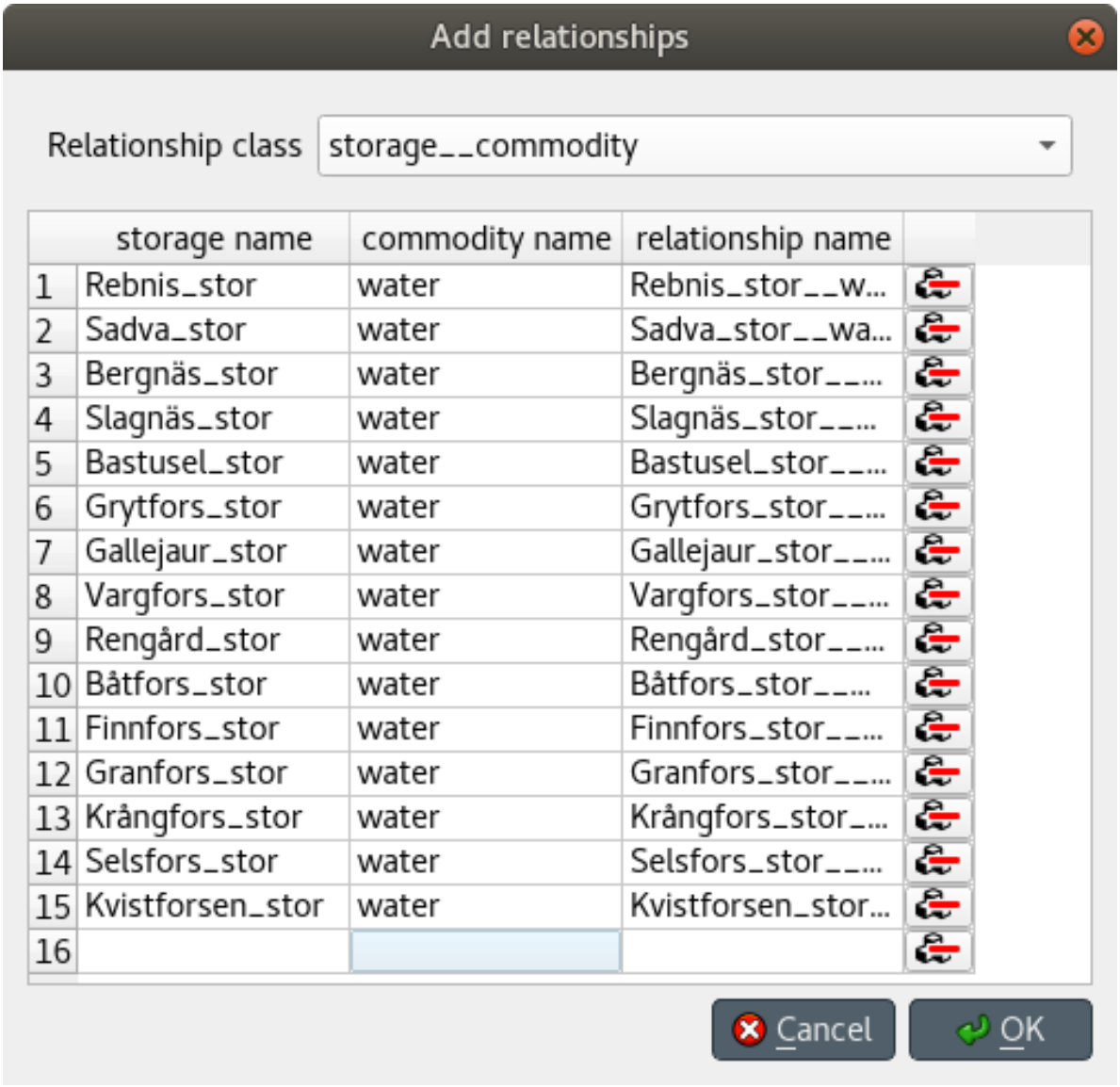
	node name	temporal_block name	relationship name
1	Rebnis_upper	some_week	Rebnis_upper___...
2	Sadva_upper	some_week	Sadva_upper__s...
3	Bergnäs_upper	some_week	Bergnäs_upper_...
4	Slagnäs_upper	some_week	Slagnäs_upper_...
5	Bastusel_upper	some_week	Bastusel_upper...
6	Grytfors_upper	some_week	Grytfors_upper_...
7	Gallejaur_upper	some_week	Gallejaur_upper...
8	Vargfors_upper	some_week	Vargfors_upper...
9	Rengård_upper	some_week	Rengård_upper_...
10	Båtfors_upper	some_week	Båtfors_upper_...
11	Finnfors_upper	some_week	Finnfors_upper_...
12	Granfors_upper	some_week	Granfors_upper...
13	Krångfors_upper	some_week	Krångfors_uppe...
14	Selsfors_upper	some_week	Selsfors_upper_...
15	Kvistforsen_upper	some_week	Kvistforsen_upp...
16	Rebnis_lower	some_week	Rebnis_lower___...
17	Sadva_lower	some_week	Sadva_lower__s...
18	Bergnäs_lower	some_week	Bergnäs_lower_...
19	Slagnäs_lower	some_week	Slagnäs_lower_...
20	Bastusel_lower	some_week	Bastusel_lower_...
21	Grytfors_lower	some_week	Grytfors_lower_...
22	Gallejaur_lower	some_week	Gallejaur_lower...
23	Vargfors_lower	some_week	Vargfors_lower...
24	Rengård_lower	some_week	Rengård_lower_...
25	Båtfors_lower	some_week	Båtfors_lower_...
26	Finnfors_lower	some_week	Finnfors_lower_...
27	Granfors_lower	some_week	Granfors_lower...
28	Krångfors_lower	some_week	Krångfors_lowe...
29	Selsfors_lower	some_week	Selsfors_lower_...
30	Kvistforsen_lower	some_week	Kvistforsen_low...
31	electricity_node	some_week	electricity_node...
32			

Cancel OK

10. Repeat the procedure to establish the connection of each storage to the corresponding unit, as relationships of class `storage__unit`:



11. Repeat the procedure to establish that all storages store water, as relationships of class `storage__commodity`:



Specifying relationship parameter values

TODO

Running Spine model

Configuring Julia

1. Go to Spine Toolbox mainwindow and from the main menu, select **File -> Settings**. This will open the *Settings* dialog.
2. Go to the *Julia* group box and enter the path to your julia executable in the first line edit.
3. (Optional) Enter the path of a julia project that you want to use with Spine Toolbox in the second line edit. Leave blank to use julia's home project.

4. Click **Ok**.
5. From the application main menu, select **File -> Tool configuration assistant.** This will install the Spine Model package to the julia project specified above. Follow the instructions until completion.

Creating a Tool template for Spine Model

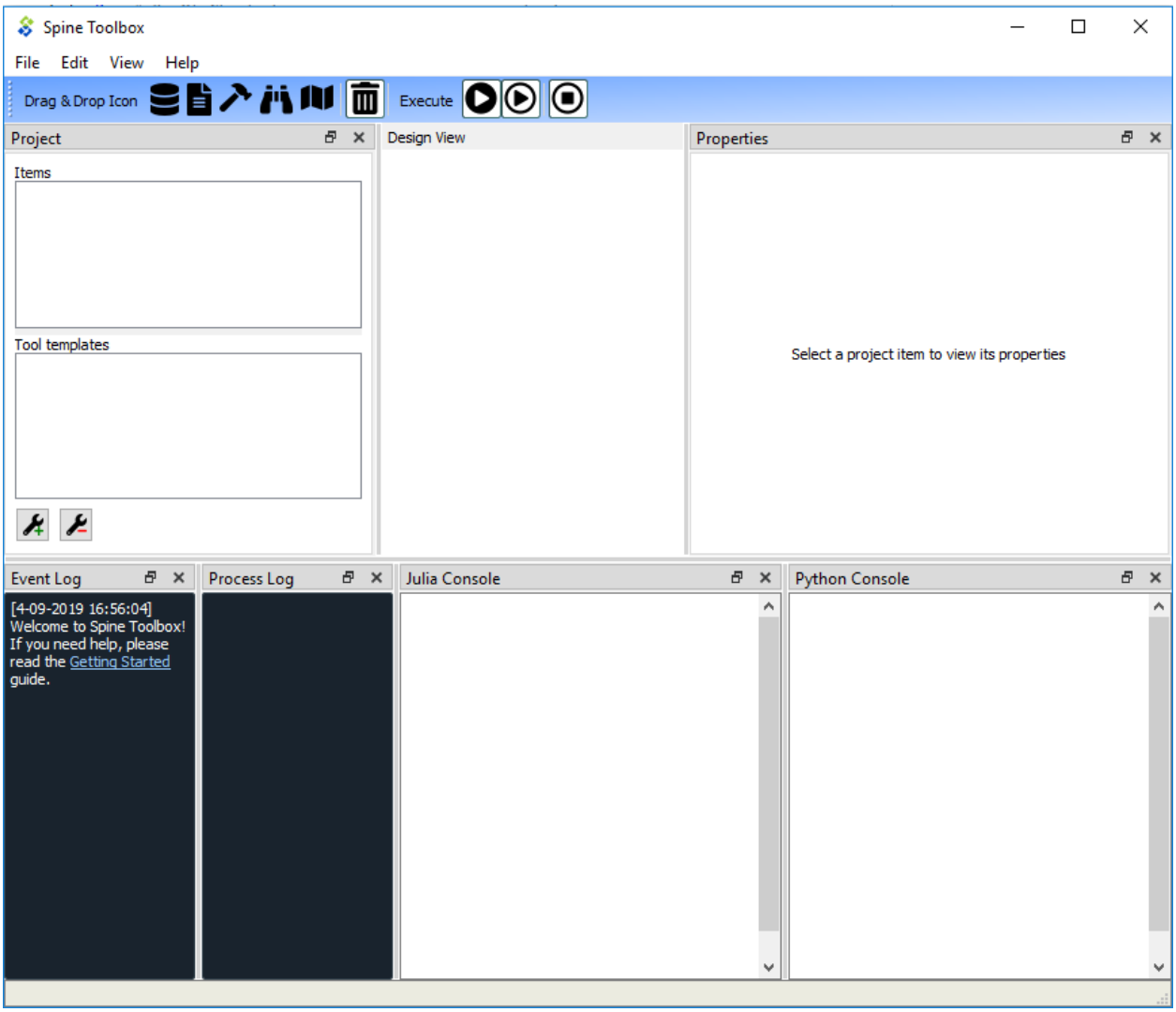
TODO

CHAPTER 3

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 templates 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, save the project, open an existing project, rename your 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

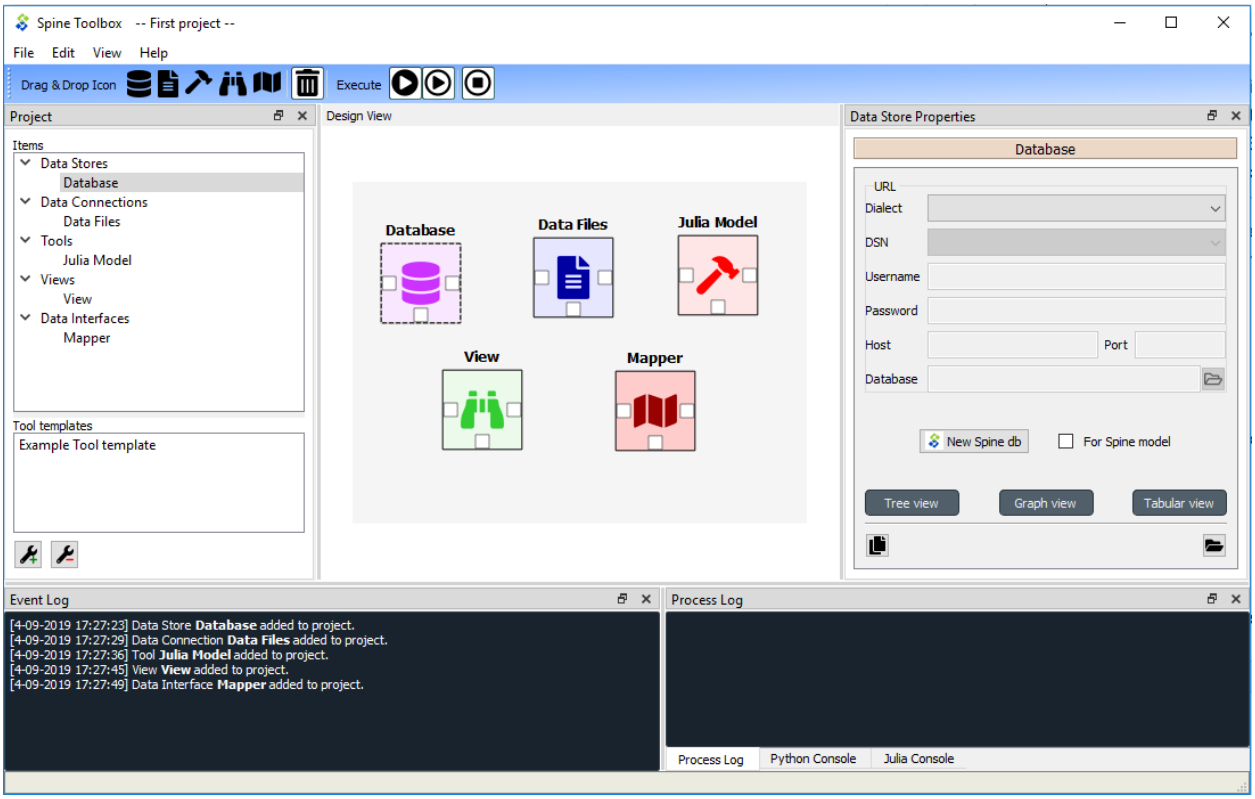
your project. The button executes all Directed-Acyclic Graphs (DAG) in the project in a row. The button executes the selected DAG. You can select a DAG to be executed by selecting a single project item from the desired DAG in the *Design View*. The button terminates DAG execution (if running).

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.

Here is one example, how you can customize the main window. In the picture, a user has created a project *First project*, which contains five project items. A Data Store called *Database*, a Data Connection called *Data Files*, A Tool called *Julia Model*, a View called *View*, and a Data Interface called *Mapper*. You can also see the project items categorized by their project item type in the *Project* dock widget.



CHAPTER 4

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 the DAG is processed by Spine Toolbox.

The following items are currently available:

4.1 Data Store

A Data store item represents a connection to a Spine model database. Currently, the item supports sqlite and mysql dialects. The database can be accessed and modified using *data store views* available from the item's properties or from a right-click context menu.

4.2 Data Connection

A Data connection item provides access to data files. It also provides access to the *Datapackage editor*.

4.3 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 or executable as well. A tool is specified by its *template*.

4.4 View

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

Note: Currently, only *Tree view* supports multiple databases.

4.5 Data Interface

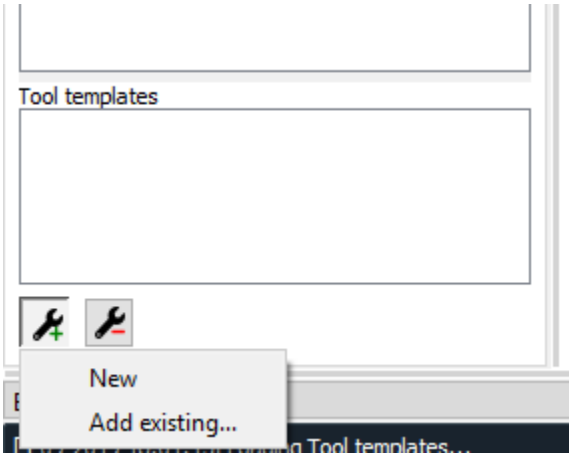
This item provides mapping from tabulated data (comma separated values, Excel) to the Spine data model.

CHAPTER 5

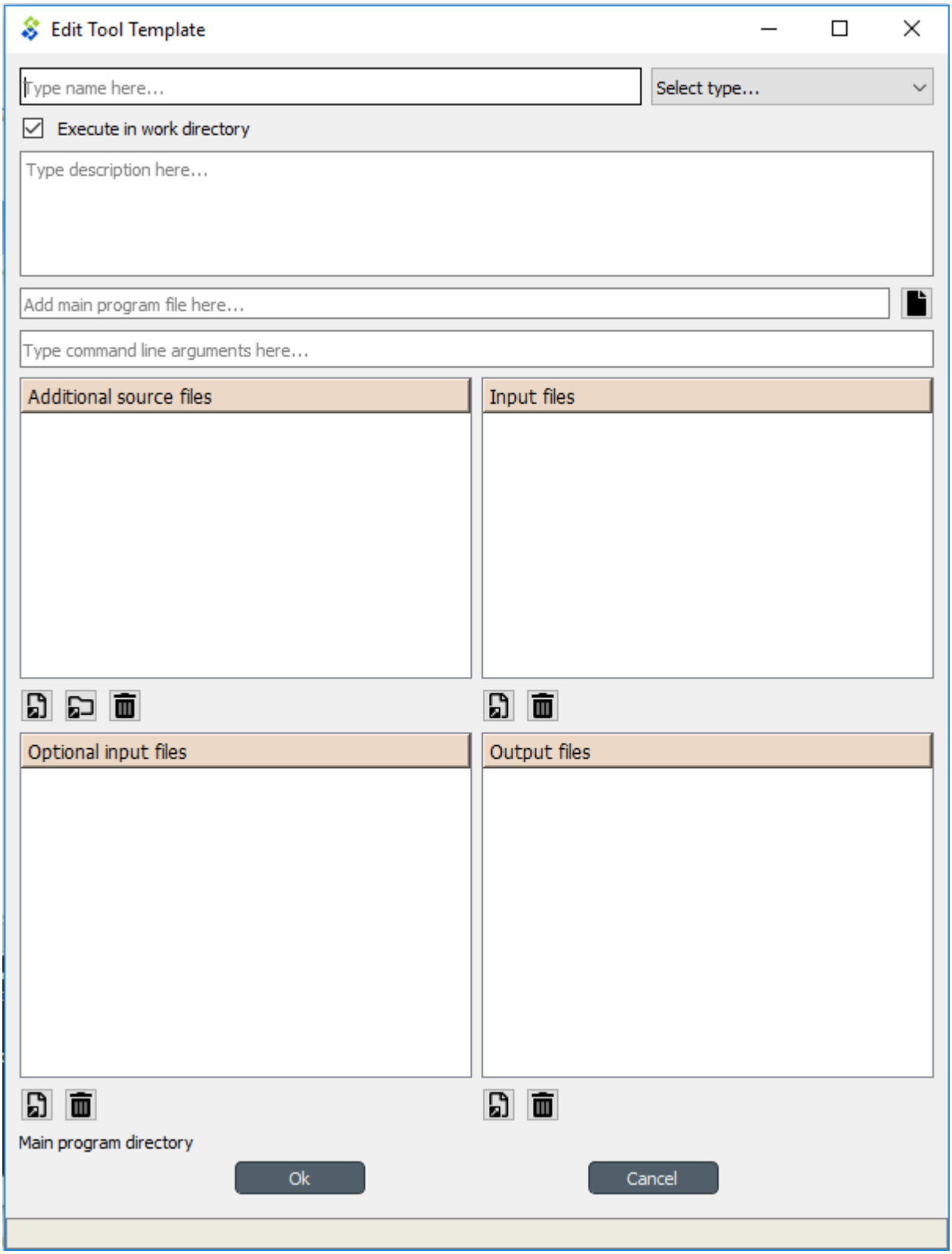
Tool template editor

This section describes how to make a new Tool template and how to edit existing Tool templates.

To execute a Julia, Python, GAMS, or an executable script in Spine Toolbox, you must first create a Tool template to your project. You can open the Tool template editor in several ways. One way is to press the Tool icon with a plus button in the *Project* dock widget. This presents you a pop-up menu with the *New* and *Add existing...* options.



When you click on *New* the following form pops up.



Start by giving the Tool template 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 template to be executed in the work directory or in its source directory (See *Terminology* section). You can give the Tool template a description, describing what the Tool

template does. Main program file is the main file of your simulation model, or an executable script. You can create a blank file into a new directory by pressing the button and selecting *Make new main program* or you can browse to find an existing main program file by pressing the same button and selecting *Select existing main program*. Command line arguments will be appended to the actual command that Spine Toolbox executes in the background, e.g. if you have a Windows batch file called *do_things.bat*, which accepts command line arguments *a* and *b*. If you set *a b* on the command line arguments. This 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 template has finished execution. Unix-style wildcards *?* and * are supported.

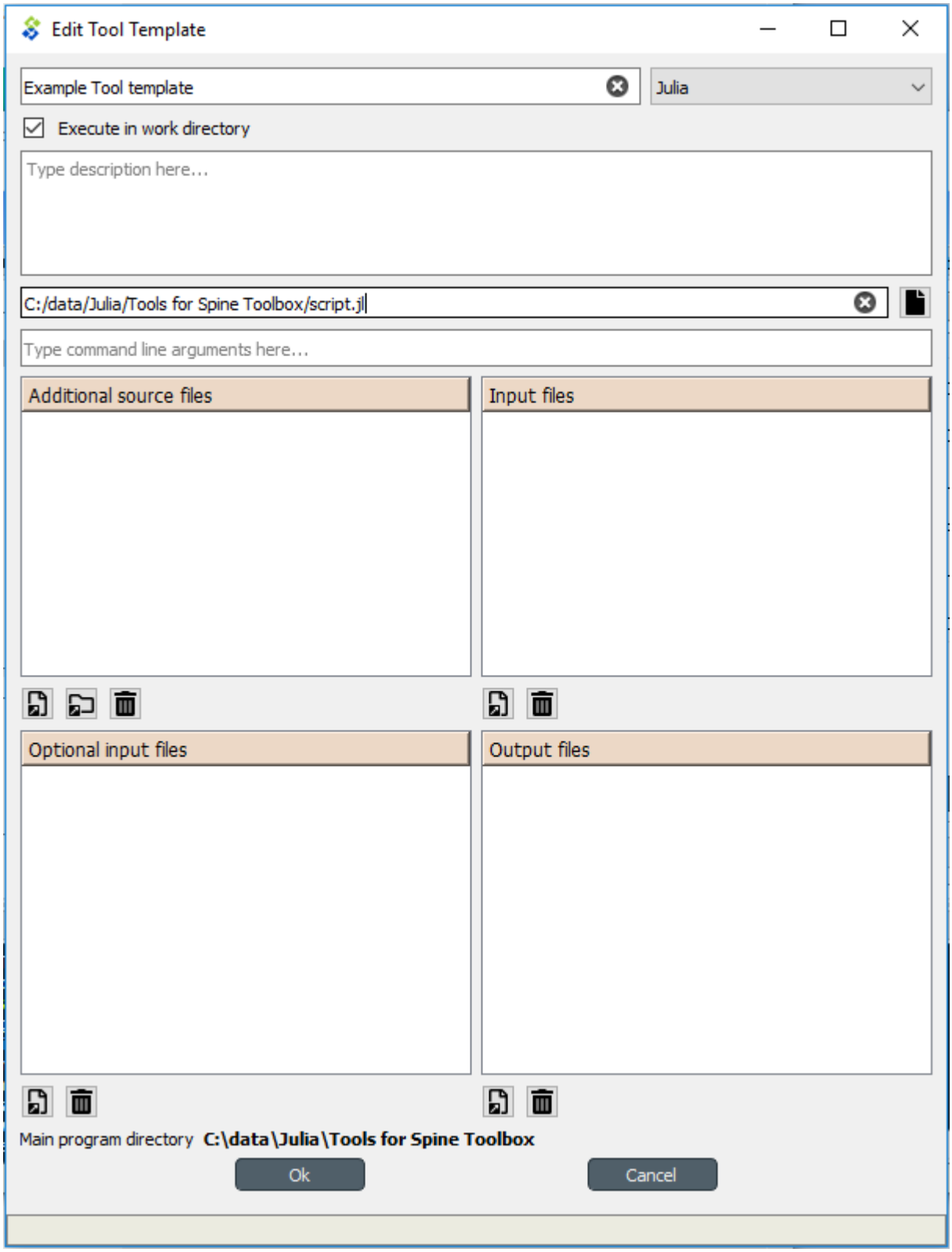
Examples:

- **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 template, click Ok, and you will be asked where to save the Tool template file. It is recommended to save the file into the same directory where the main program file is located. The Tool template file is a text file in JSON format and has an extension *.json*

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

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

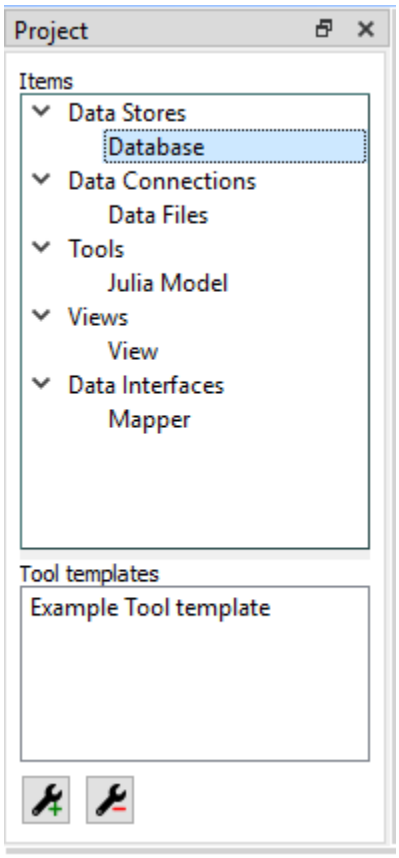


Note: Under the hood, the contents of the Tool template are saved to a *Tool template definition file* in JSON format. Users do not need to worry about the contents of these files since reading and writing them is managed by the app. For

the interested, here are the contents of the *Tool template definition file* that we just created.:

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

After the user has clicked Ok and saved the file, the new Tool template has been added to the project.



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

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

CHAPTER 6

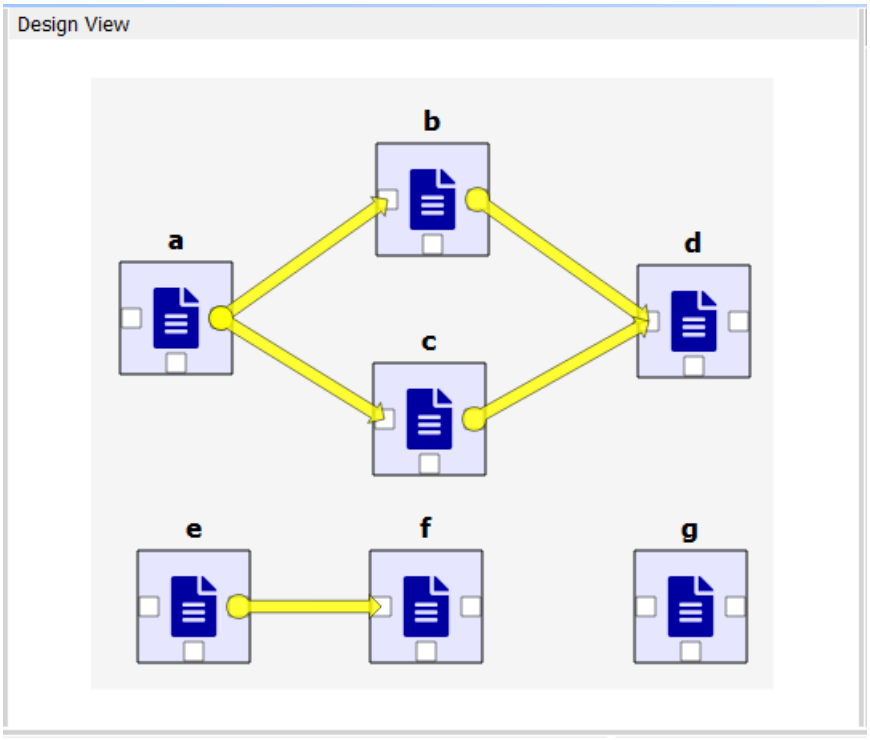
Executing Projects

This section describes how executing a project works. Execution happens by pressing the or the 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 edges is a DAG.
2. All project items that are connected, are considered as a single DAG. It does not matter, which direction the edges go. If there is a path between two nodes, 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: nodes: a, b, c, d. edges: a-b, a-c, b-d, c-d
- DAG 2: nodes: e, f. edges: e-f
- DAG 3: nodes: g. edges: None

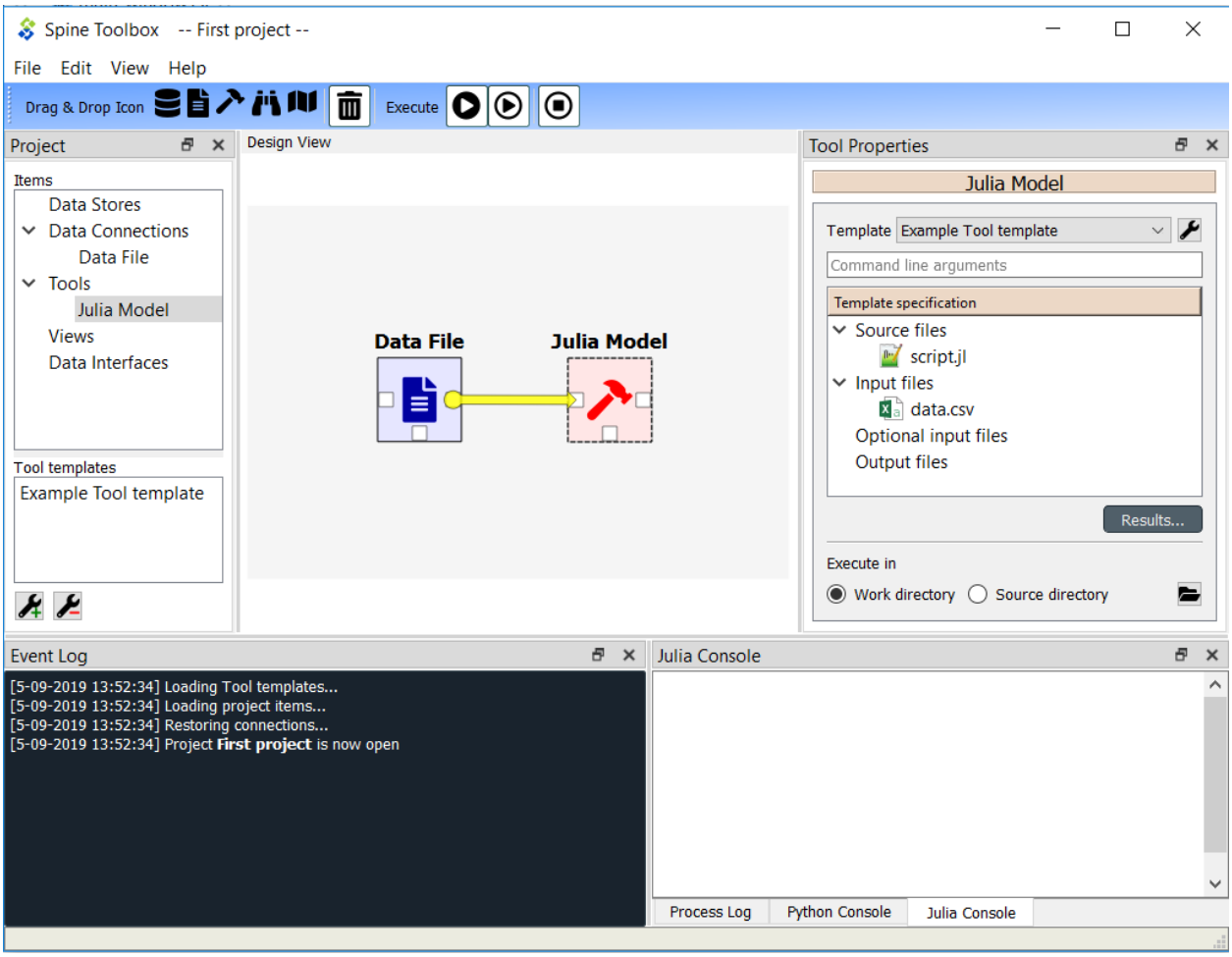
When you press the button, all three DAGs are executed in a row. You can see the progress and the current executed node in the *Event Log*. Execution order of DAG 1 is *a->b->c->d* or *a->c->b->d* since nodes b and c are siblings. DAG 2 execution order is *e->f* and DAG 3 is just *g*. If you have a DAG in your project that breaks the rules above, that DAG is skipped and the execution continues with the next DAG.

You can execute a single DAG in the project by selecting a node in the desired DAG and pressing the button in the tool bar. For example, to execute only DAG 1 you can select (in *Design View* or in the project item list in *Project* dock widget) any of the nodes a, b, c, or d and then press the button.

Tip: If you are not sure how execution works in your DAG, you can test the execution just like in the above picture by adding and connecting empty Data Connection items and then pressing the play buttons.

6.1 Executing Tools as a part of the DAG

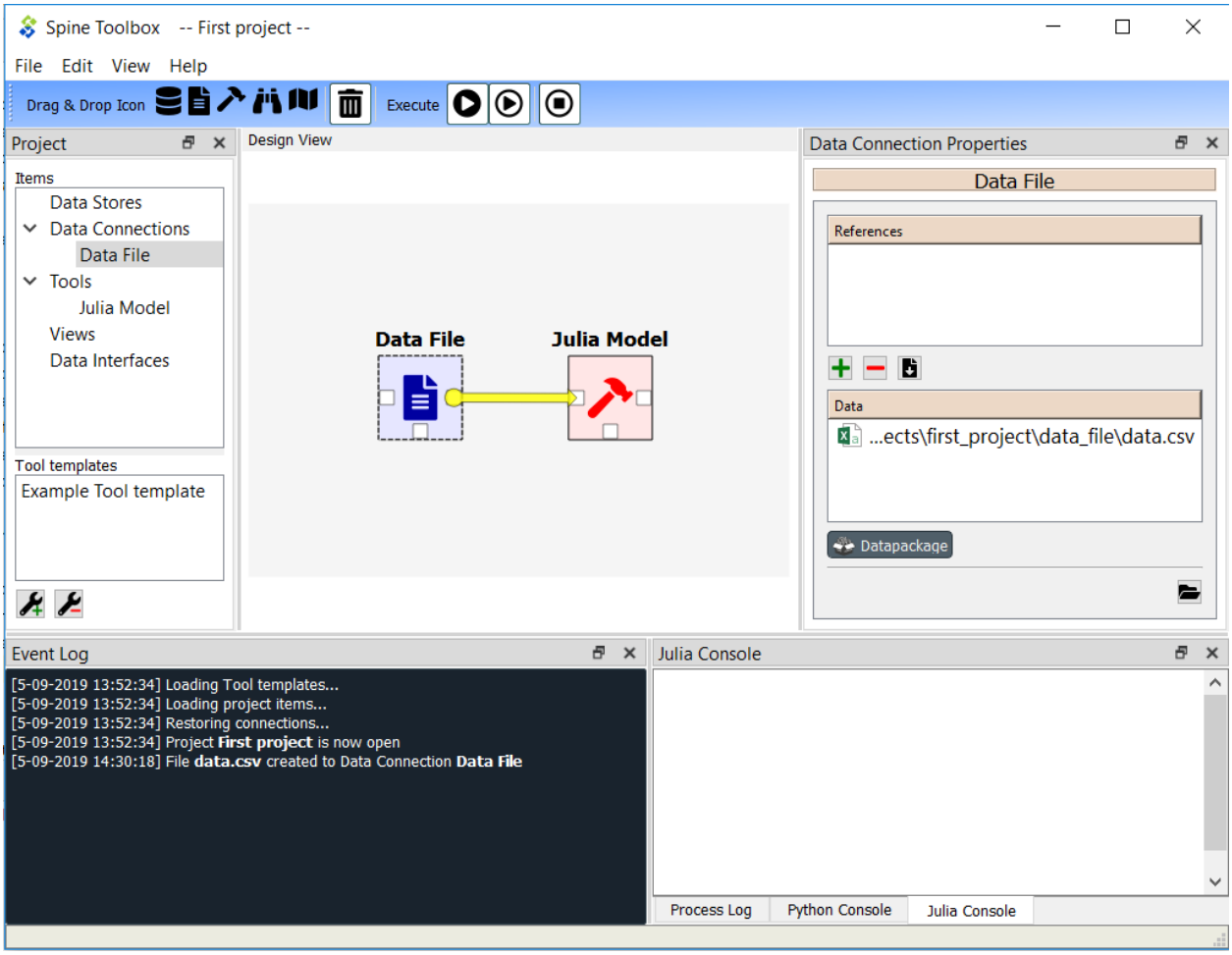
All project items in the DAG are *executed*, but the real processing only happens when a Tool project item is executed. When you have created at least one Tool template, you can execute a Tool as part of the DAG. The Tool template defines the process that is depicted by the Tool project item. As an example, below we have two project items; *Julia Model* Tool and *Data File* Data Connection connected as a DAG with an edge between the nodes.



Selecting the *Julia Model* shows its properties in the *Properties* dock widget. In the top of the Tool Properties, there is a template drop-down menu. From this drop-down menu, you can select the Tool template for this particular Tool item. The *Example Tool Template* tool template has been selected for the Tool *Julia Model*. Below the drop-down menu, you can see the details of the Tool template, command line arguments, Source files (the first one is the main program file), Input files, Optional input files and Output files. *Results...* button opens the Tool's result archive directory in the File Explorer (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.

Note: The *Example Tool Template* has been modified a bit from previous sections. It now requires an *Input file data.csv*.

When you click on the button, the execution starts from the *Data File* Data Connection. When executed, Data Connection items *advertise* their files and references to project items that are in the same DAG and executed after them. In this particular example, the *Data File* item contains a file called *data.csv* as depicted in the below picture.

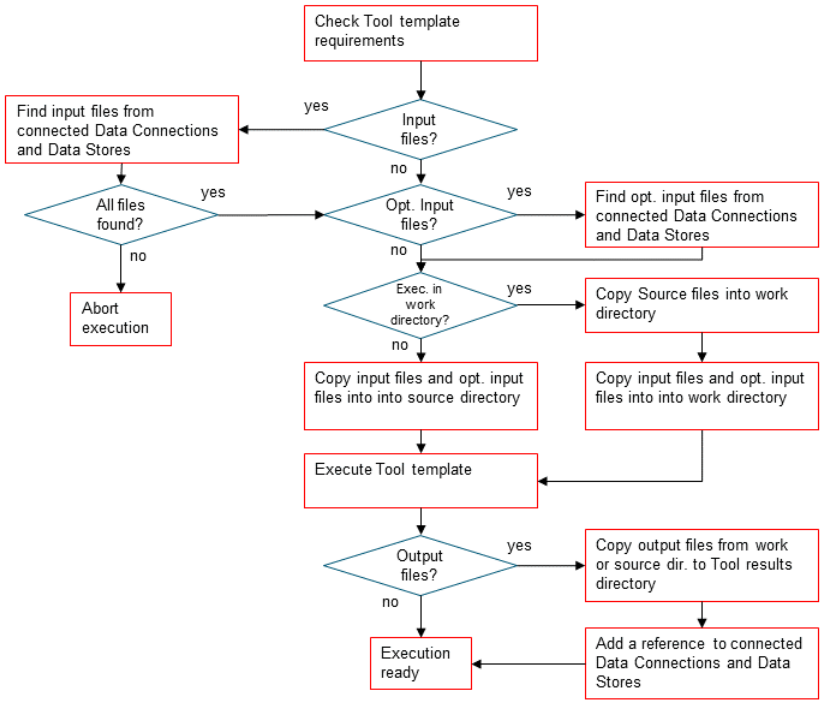


When it's the *Julia Model* tools turn to be executed, it checks if it finds the file *data.csv* from project items, that have already been executed. When the DAG is set up like this, the Tool finds the input file that it requires and then starts processing the Tool template starting with the main program file *script.jl*. Note that if the edge 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 file *data.csv*. The same thing happens if there is no edge between the two project items. In this case the project items would be in separate DAGs.

Since the Tool template 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).

6.2 Tool execution algorithm

The below figure depicts what happens when a Tool item with a valid Tool template is executed.


Check Tool template requirements
Input files?
yes
Find input files from connected Data Connections and Data Stores
no
All files found?
yes
no
Abort execution
Opt. Input files?
yes
Find opt. input files from connected Data Connections and Data Stores
no
Exec. in work directory?
yes
Copy Source files into work directory
no
Copy input files and opt. input files into into source directory
Copy input files and opt. input files into into work directory
Execute Tool template
Output files?
yes
Copy output files from work or source dir. to Tool results directory
no
Execution ready
Add a reference to connected Data Connections and Data Stores

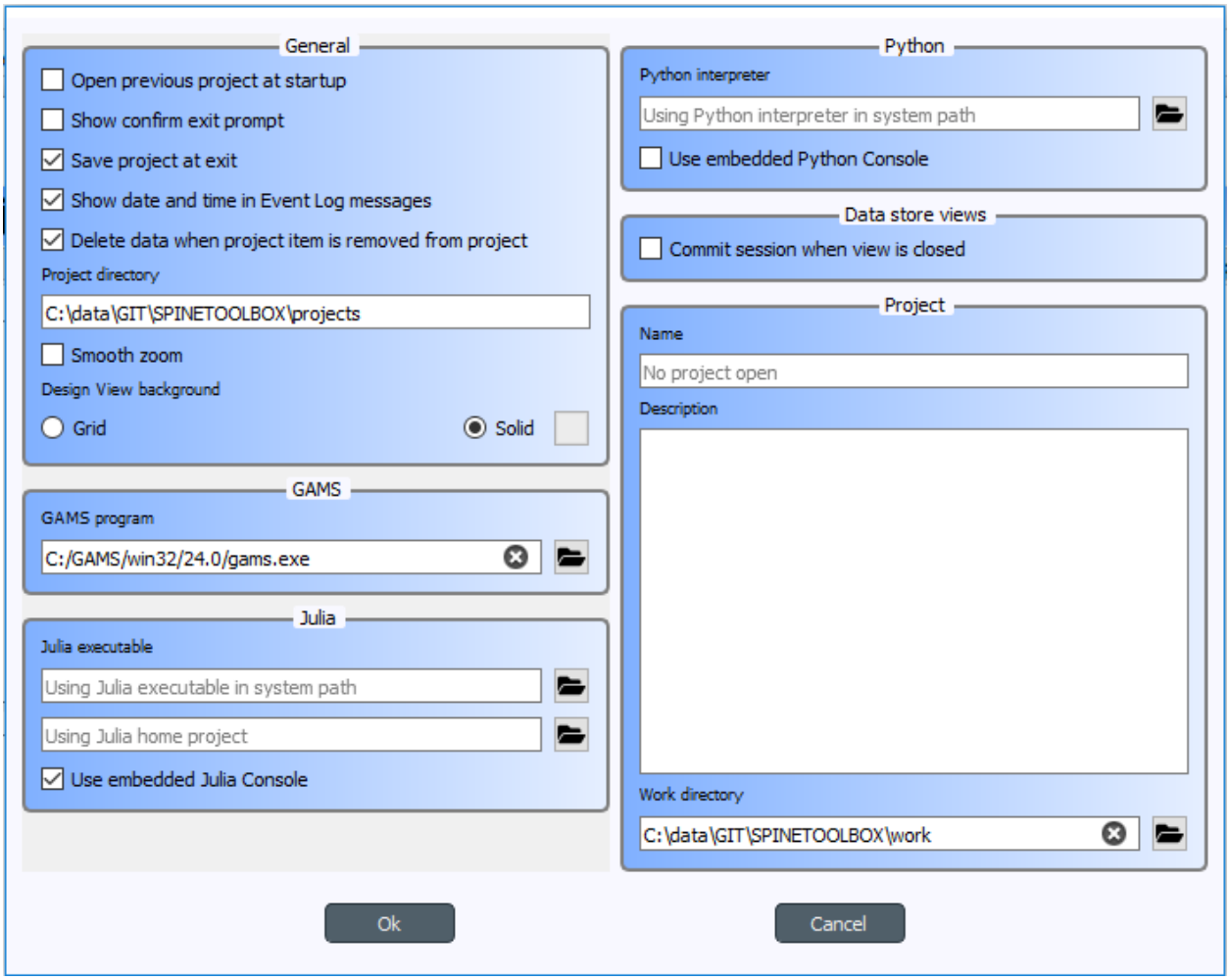
CHAPTER 7

Settings

Spine Toolbox settings are categorized in the following way

7.1 Application settings

You can open the application settings from the main window menu *File->Settings...*, or by pressing *F1*.



The settings on this form have been categorized into six categories. *General*, *GAMS*, *Julia*, *Python* and *Data store views* settings are general application settings, which affect all projects. Settings in the *Project* category only affect the current project.

7.1.1 General 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, date and time is appended into every Event Log message.
- **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.
- **Project directory** Directory where projects are saved. This is non-editable at the moment.

- **Smooth zoom** Controls the way zooming (by using the mouse wheel) behaves in Design View and Graph View. Controls if the zoom in/out is continuous or discrete. On older computers, smooth zoom is not recommended.
- **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.

7.1.2 GAMS settings

- **GAMS program** Path to Gams executable you wish to use to execute GAMS Tool templates. Leave this blank to use the system GAMS i.e. GAMS set up in your system PATH variable.

7.1.3 Julia settings

- **Julia executable** Path to Julia executable you wish to use to execute Julia Tool templates. This is the Julia that will be used in the embedded Julia Console and also the Julia that is used when executing Julia Tool templates as in the shell. Leave this blank, if you wish to use the system Julia.
- **Julia home project** Set the Julia home project here.
- **Use embedded Julia Console** Check this box to execute Julia Tool templates in the built-in Julia Console. If you leave this un-checked, Julia Tool templates will be executed as in the shell. For example, on Windows this would be the equivalent as 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. It is highly recommended to use the embedded Julia Console, since this gives significant performance improvements compared to shell execution.

7.1.4 Python settings

- **Python interpreter** Path to Python executable you wish to use to execute Python Tool templates. This is the Python that will be used in the embedded Python Console and also the Python that is used when executing Python Tool templates as in the shell. Leave this blank, if you wish to use the system Python.
- **Use embedded Python Console** Check this box to execute Python Tool templates in the embedded iPython Console. If you un-check this box, Python Tool templates will be executed as in the shell. For example, on Windows this would be the equivalent as running command `python.exe script.py` in the command prompt. If you decide to use the embedded Python Console, the `script.py` is executed there.

7.1.5 Data Store views settings

- **Commit session when view is closed** This checkbox controls what happens when you close the Tree view, the Graph view, or the tabular view and when you have uncommitted changes. Unchecked: Does not commit session and does not show message box. Partially checked: Shows message box (default). Checked: Commits session and does not show message box.

7.1.6 Project settings

These settings affect only the project that is currently open.

- **Name** Current project name. If you want to change the name of the project, use menu option *File-Save as...*.
- **Description** Current project description. You can edit the description here.

- **Work directory** Directory where processing the Tool takes place. You can change this directory. Make sure to clean up the directory every now and then.

7.2 Project item settings / properties

Each project item (Data Store, Data Connection, Tool, View, and Data Interface) has its own set of properties. These are saved into the project save file. You can view and edit them in project item properties on the main window.

7.3 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).

7.4 Where are the Settings stored?

Application Settings and Preferences are saved to a location that depends on your operating system. On Windows, there is no separate settings file, the settings are stored into registry key `HKEY_CURRENT_USER\Software\SpineProject\Spine Toolbox`. It is safe to delete this key if you want to reset application to factory settings.

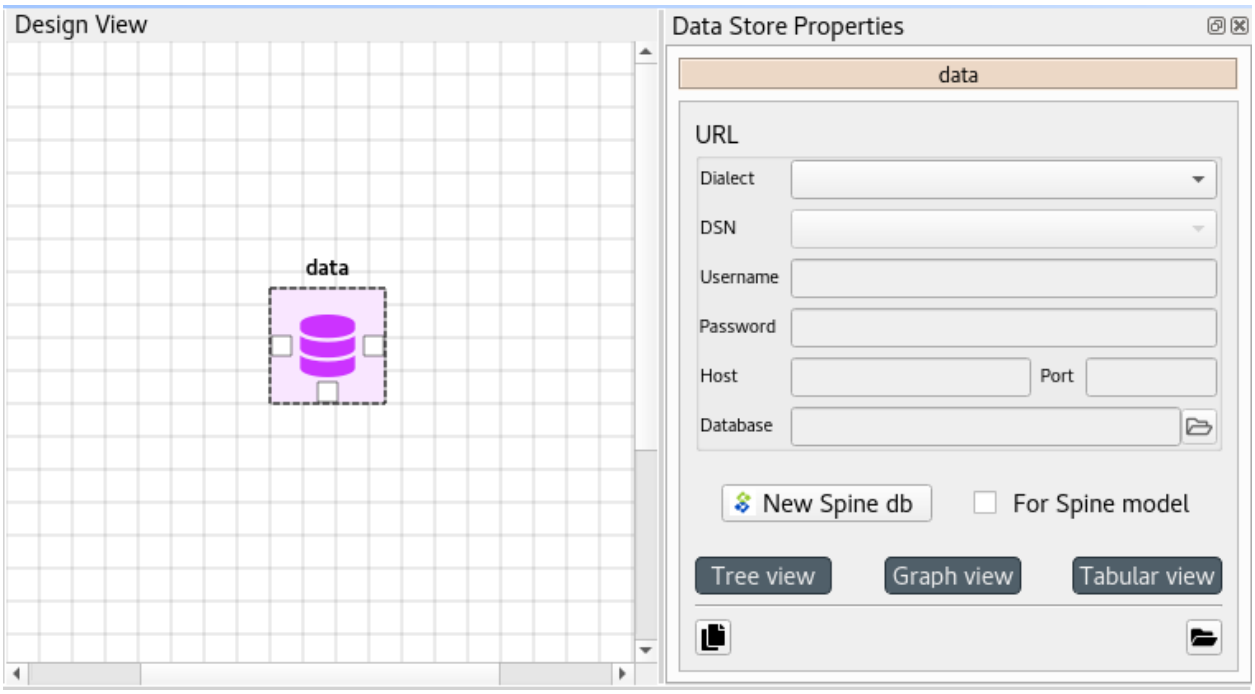
Projects are saved to *.proj* files in the Project directory. In addition, each project has its own dedicated directory under the Project directory which can be used to keep data from different projects separate. All project items in a project have their own directory under that project's directory, where individual project item data can be stored (e.g. *.sqlite* files in Data Store directories).

CHAPTER 8

Data store views

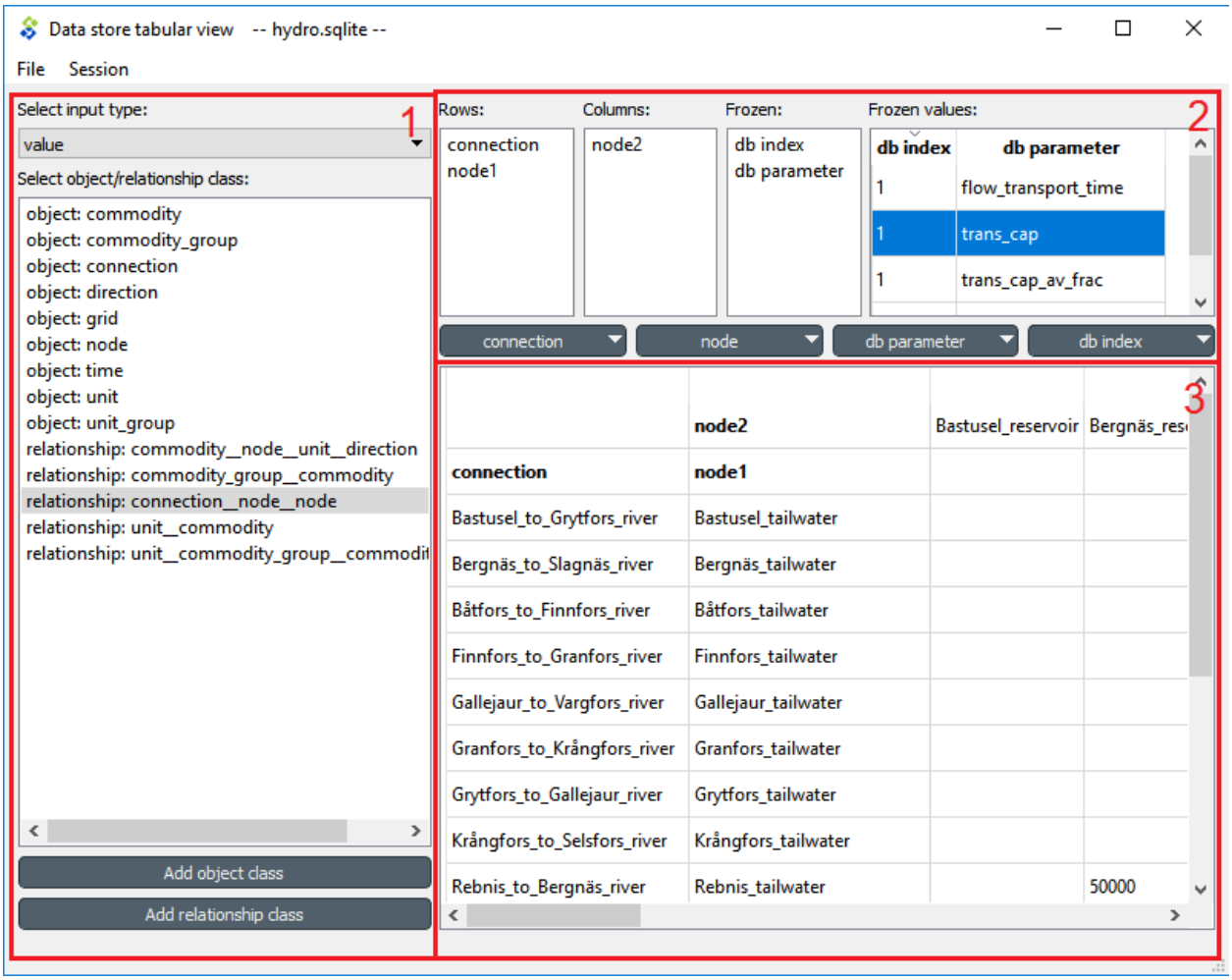
This section describes the different interfaces for viewing and editing data in a Spine database.

To open any of the viewing interfaces, select a **Data Store** and click the corresponding button in its *Properties*:



8.1 Tabular view

The **Tabular view** is used to display and edit the data in a Spine database via a table-like interface. The interface lets you filter and pivot the data for exploration and editing. To start the **Tabular view**, select a **Data Store** item and press the **Tabular view** button in its *Properties*.



The **Tabular view** has three main components:

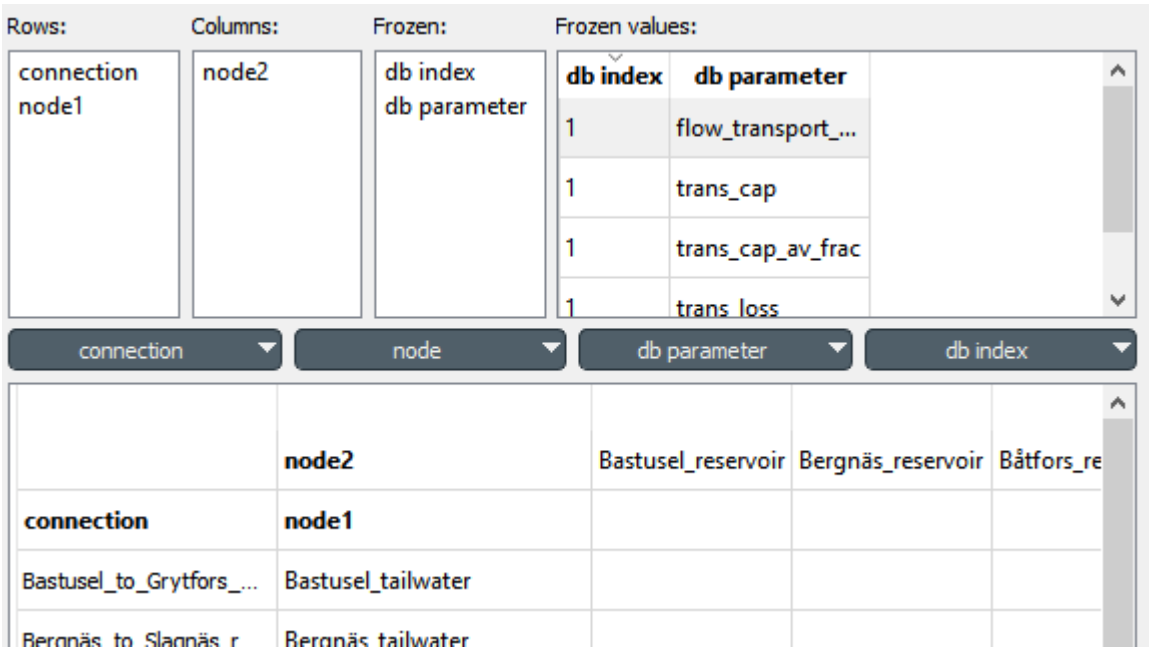
- *Selection component* (1), where you can select the object class or relationship class to visualize.
- *Pivot lists* (2), where you can transform the data view and choose, e.g., which items go into rows and which into columns.
- *Main table* (3), where the actual data is displayed.

From the drop-down list at the top of the selection component, you can select two different input types:

- *value*: display all objects (or relationships), as well as all parameters and parameter values for the selected object (or relationship) class.
- *set*: display only the objects (or relationships) for the selected object (or relationship) class.

8.1.1 Pivoting and filtering data

You can transform (pivot) the data-view by dragging items across the *Pivot lists*:



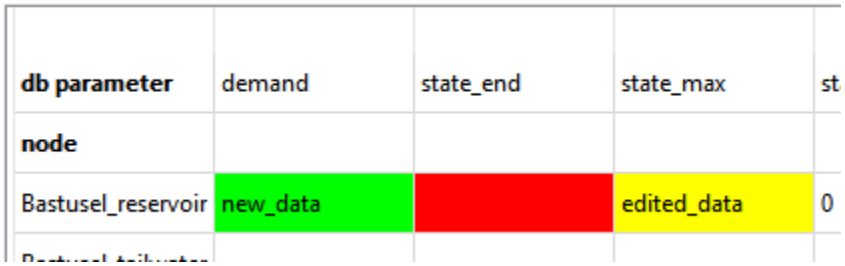
Here is how each of the *Pivot lists* works:

- *Rows*: All items in this list are displayed in the *Main table* so there is a unique row for each object (or relationship) that belongs to that object (or relationship) class.
- *Columns*: All items in this list are displayed in the *Main table* so there is a unique column for each object (or relationship) that belongs to that object (or relationship) class.
- *Frozen*: All items in this list are excluded from the *Main table* and shown in the *Frozen values* table instead; the *Main table* is then filtered by the selected item in the *Frozen values* table.

To filter a specific item you can use the filter buttons just above the *Main view*. You can apply multiple filters at the same time.

8.1.2 Editing Data

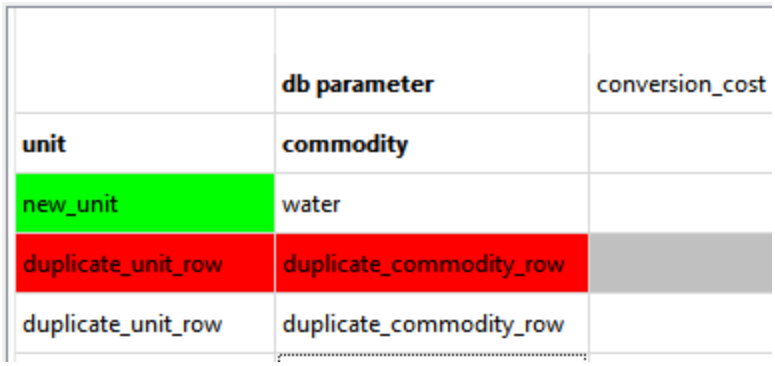
When editing parameter *values*, cells get painted in different colors:



- Green: New data
- Yellow: Edited data
- Red: Deleted data

To restore a cell to its initial value, right click on it and select **Restore value** from the context menu. You can also hover an edited cell to see the original value.

When editing item *names*, cells also get painted in different colors although their meaning is a bit different:



	db parameter	conversion_cost
unit	**commodity**	
new_unit	water	
duplicate_unit_row	duplicate_commodity_row	
duplicate_unit_row	duplicate_commodity_row	

- Green: New item. This means that all objects and parameters in green cells will be inserted when committing changes.
- Red: Invalid item. This means that the item, as well as all data for that row/column cannot be inserted when committing changes. The affected rows/columns will get a gray background. Invalid names are:
- Empty names: An item must have a name.
- Duplicate names: If the name (or combination of item names) is already assigned to an object (or relationship).
- Existing name: If the name is already taken by another object or parameter.

If you edit an item's name, the original item is not deleted from the database on commit. To delete an item from the database, right click on the cell with the name, and select **Delete item:name** from the context menu.

A new relationship is added as soon as a valid combination of objects for that relationship class is entered, even if the row/column is invalid. To remove a relationship, right click on it and select **Delete item:name** from the context menu.

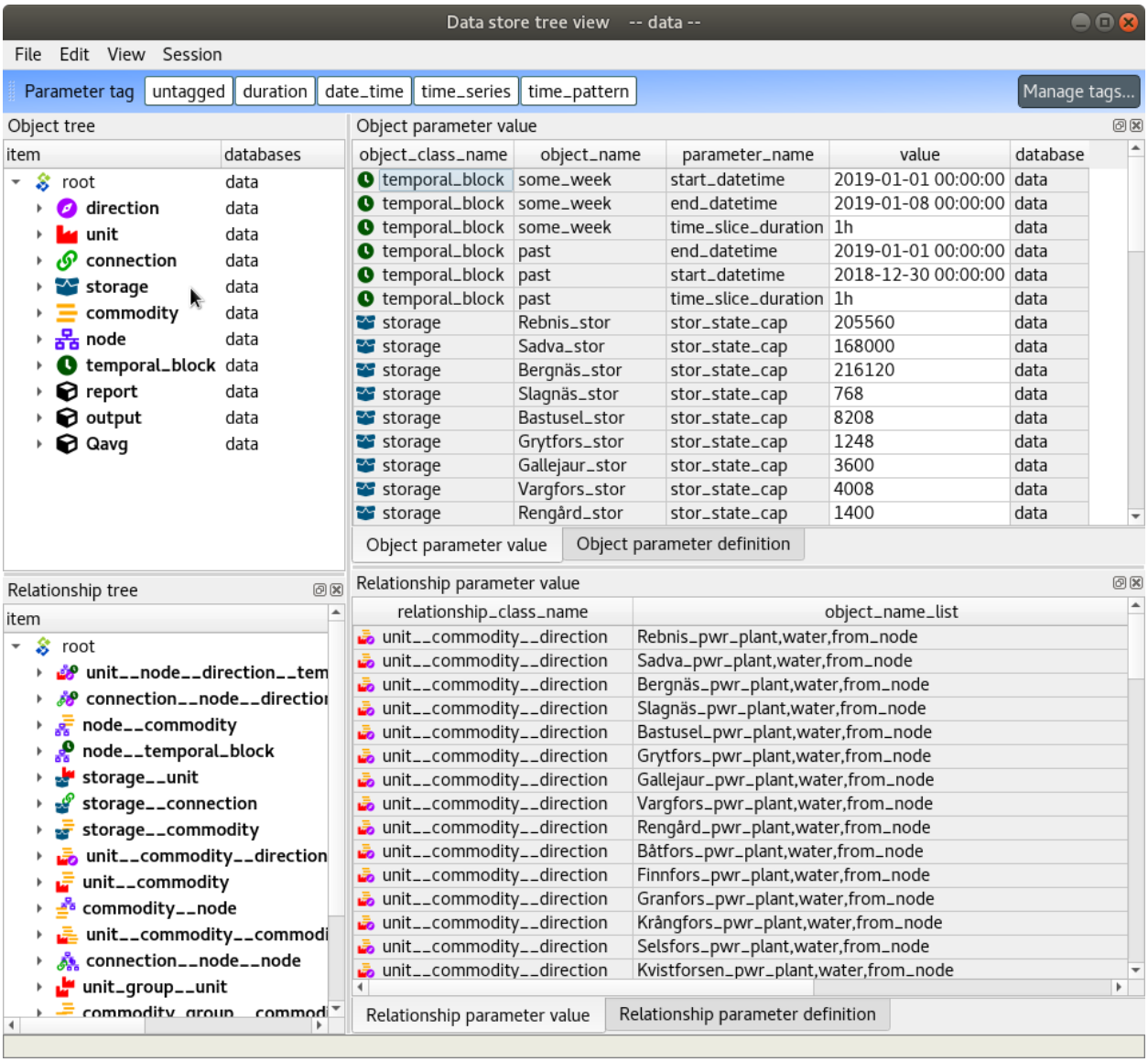
8.1.3 Commit/Rollback changes

To save changes select **Session -> Commit** from the menu bar, enter a commit message and press **Commit**. Any changes made in the **Tabular view** will be saved into the database.

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

8.2 Tree view

The **Tree view** is used to display the different object and relationship classes, with their objects and relationships in a hierarchical tree. You can also add, edit, and delete object classes, relationship classes, objects, relationships, parameters and parameter values. The **Tree view** is useful to get an overview of the data and the relationships within a Spine database:

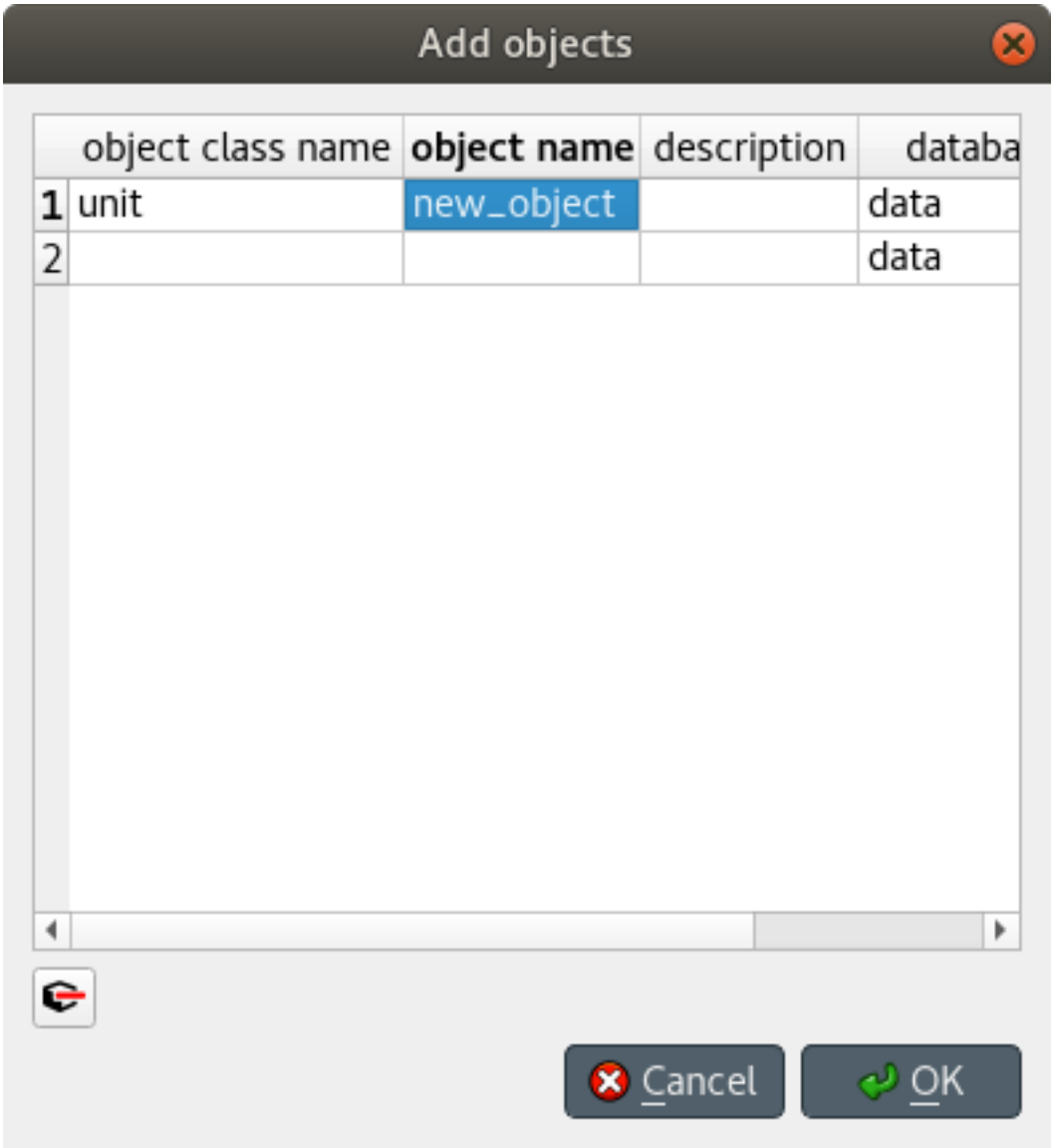


The interface has four main components:

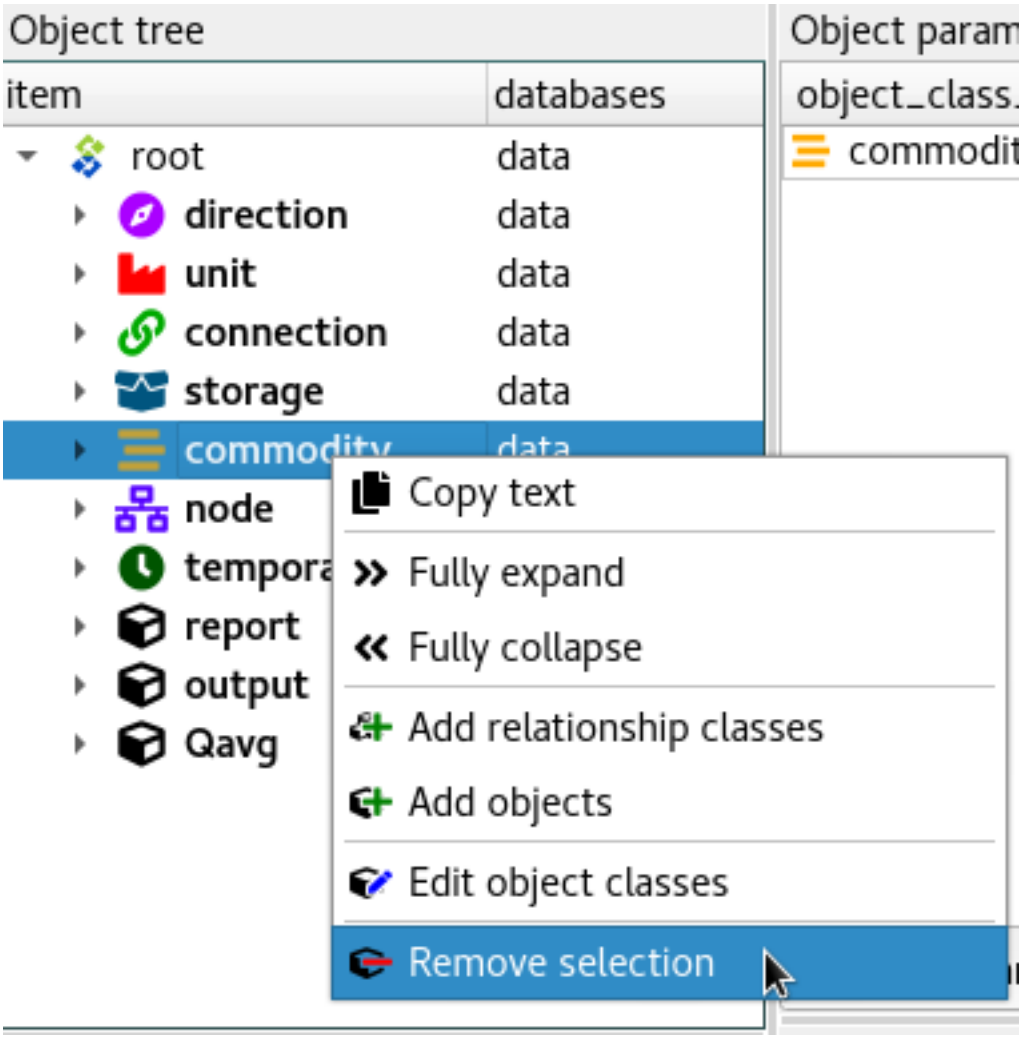
1. *Object tree*, where you can expand and collapse the different levels of the hierarchy. It also acts as a filtering tool for the other two table components, so that only items selected in the *Object tree* are shown in the *Parameter tables*.
2. *Relationship tree*, similar to *Object tree* but for relationships.
3. *Object parameter table*, where you can view, add, edit, and delete object parameter definitions and values.
4. *Relationship parameter table*, where you can view, add, edit, and delete relationship parameter definitions and values.

8.2.1 Editing items

To add object classes, relationship classes, objects or relationships you can use the **Edit** menu from the main menu bar, as well as the context menu from the *Object tree*. In the dialog that pops up you can enter new items by typing their names or pasting data from the clipboard.



To delete an item, you can again use the **Edit** menu from the main menu bar or the item's context menu from the *Object tree*.



Editing items is done following a similar procedure.

8.2.2 Viewing parameter definitions and values

In the *Parameter tables*, you can switch between viewing parameter definitions or values by using the tabs in the upper right corner.

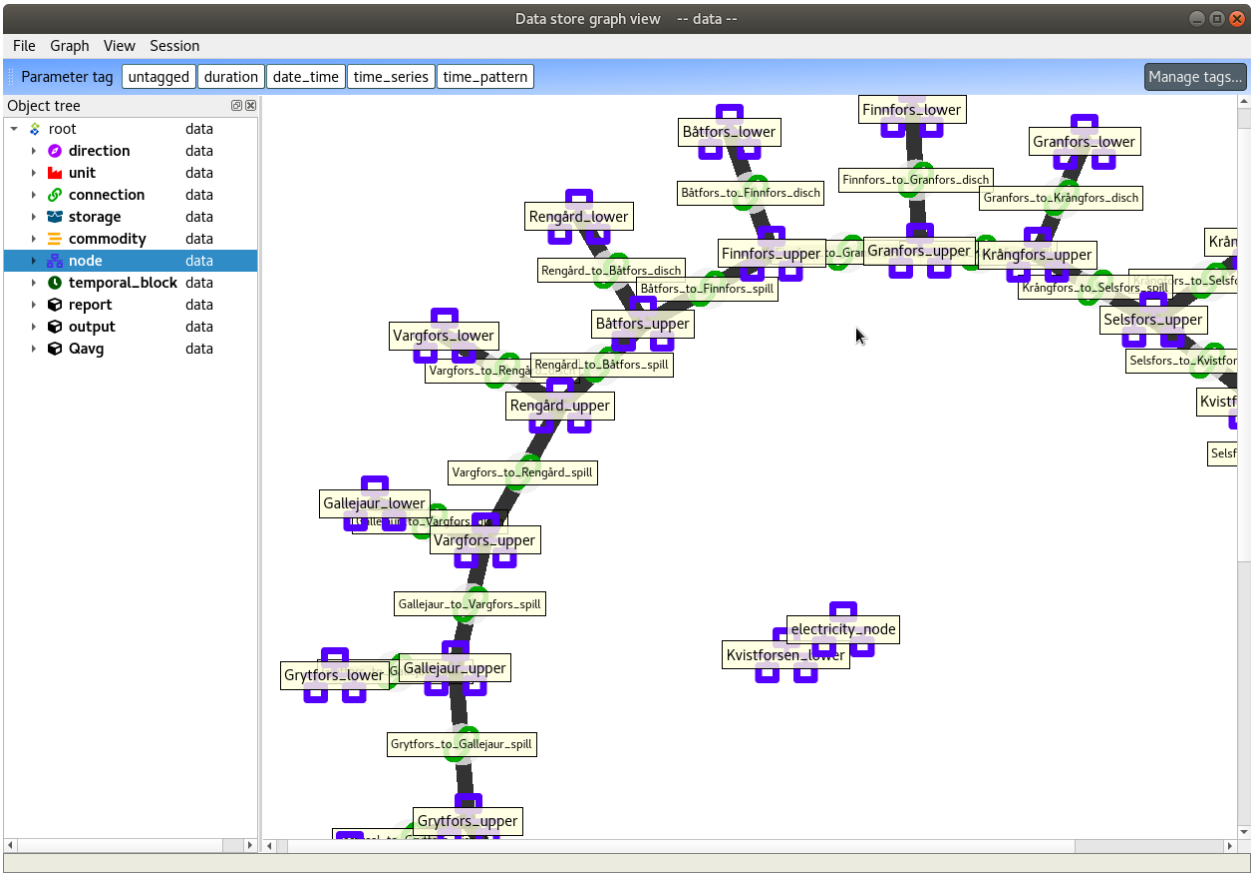
You can also (further) filter the tables by clicking on the column headers.

8.2.3 Editing parameters definitions and values

To add new parameter definitions or values you can directly do it in the last row of each table. The tables also support pasting values from the clipboard.

8.3 Graph view

The **Graph view** is used to visualize the Spine database structure into a graph. Here you can select objects to see how they are related. You can also view parameter definition and values same as in the **Tree view**.

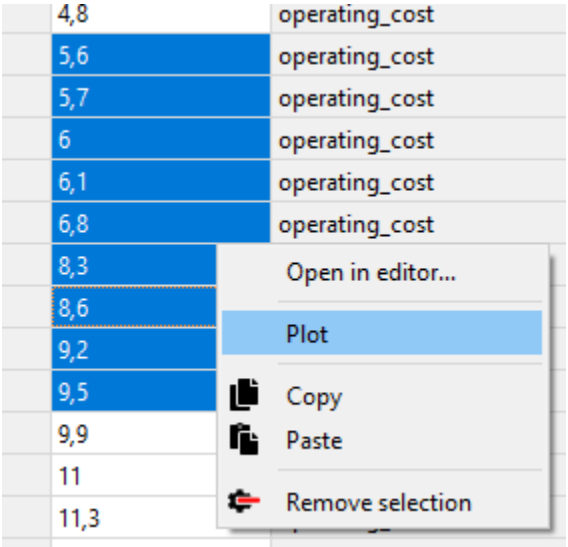

Data store graph view -- data --
File Graph View Session
Parameter tag untagged duration date_time time_series time_pattern
Manage tags...
Object tree
root data
direction data
unit data
connection data
storage data
commodity data
node data
temporal_block data
report data
output data
Qavg data
Båtfors_lower
Finnfors_lower
Granfors_lower
Båtfors_to_Finnfors_disch
Finnfors_to_Granfors_disch
Granfors_to_Krångfors_disch
Rengård_lower
Rengård_to_Båtfors_disch
Finnfors_upper
Granfors_upper
Krångfors_upper
Båtfors_to_Finnfors_spill
Krångfors_to_Selsfors_spill
Selsfors_upper
Båtfors_upper
Vargfors_lower
Selsfors_to_Kvistfor
Rengård_to_Båtfors_spill
Rengård_upper
Vargfors_to_Rengård_spill
Gallejaur_lower
Vargfors_upper
Gallejaur_to_Vargfors_spill
electricity_node
Kvistforsen_lower
Grytfors_lower
Gallejaur_upper
Grytfors_to_Gallejaur_spill
Grytfors_upper

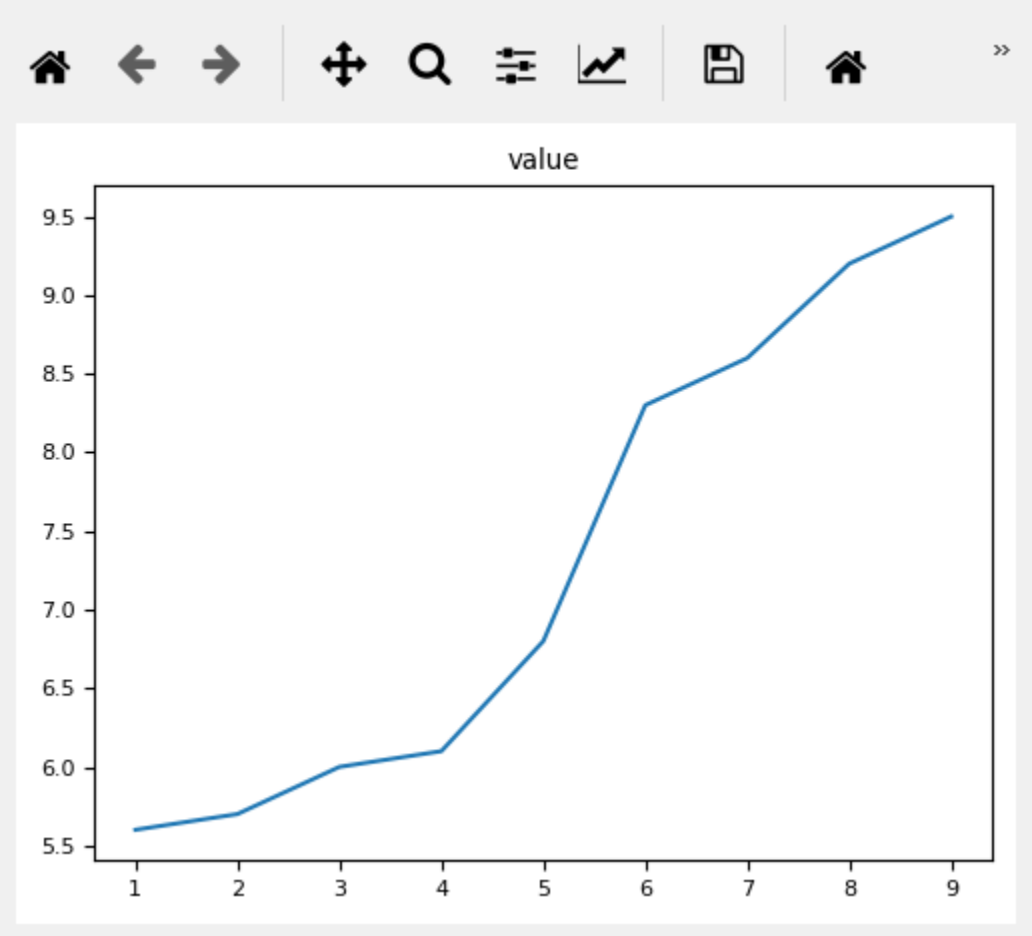
CHAPTER 9

Plotting

Basic data visualization is available in the data store views. Currently, it is possible to plot plain parameter values as well as time series. There are some limitations in plotting data from different sources, however. For instance, object and relationship parameter time series cannot be plotted on the same graph at the moment.

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



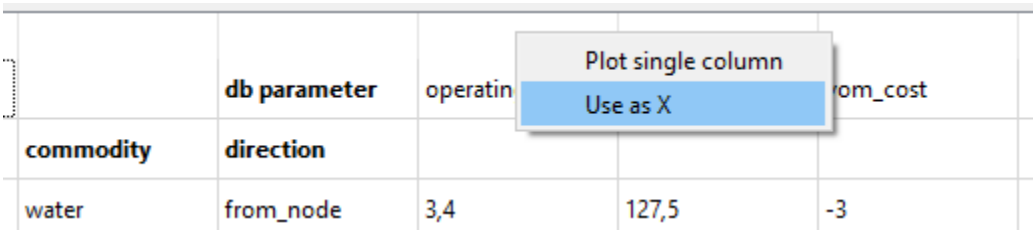


Selecting data in multiple columns plots the selection in a single window.

9.1 X axis for plain values

It is possible to plot plain values against X values given by a designated column in the pivot table of the Tabular view.

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. An *(X)* in the topmost cell indicates that the column is designated as containing 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.

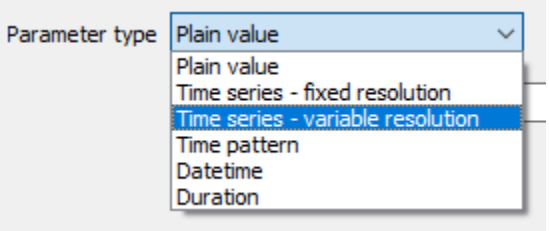
CHAPTER 10

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 like 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 data store views.

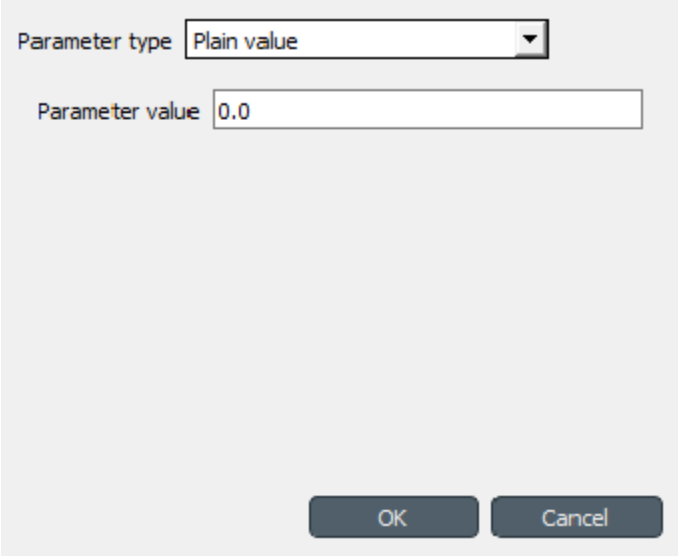
10.1 Choosing value type



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

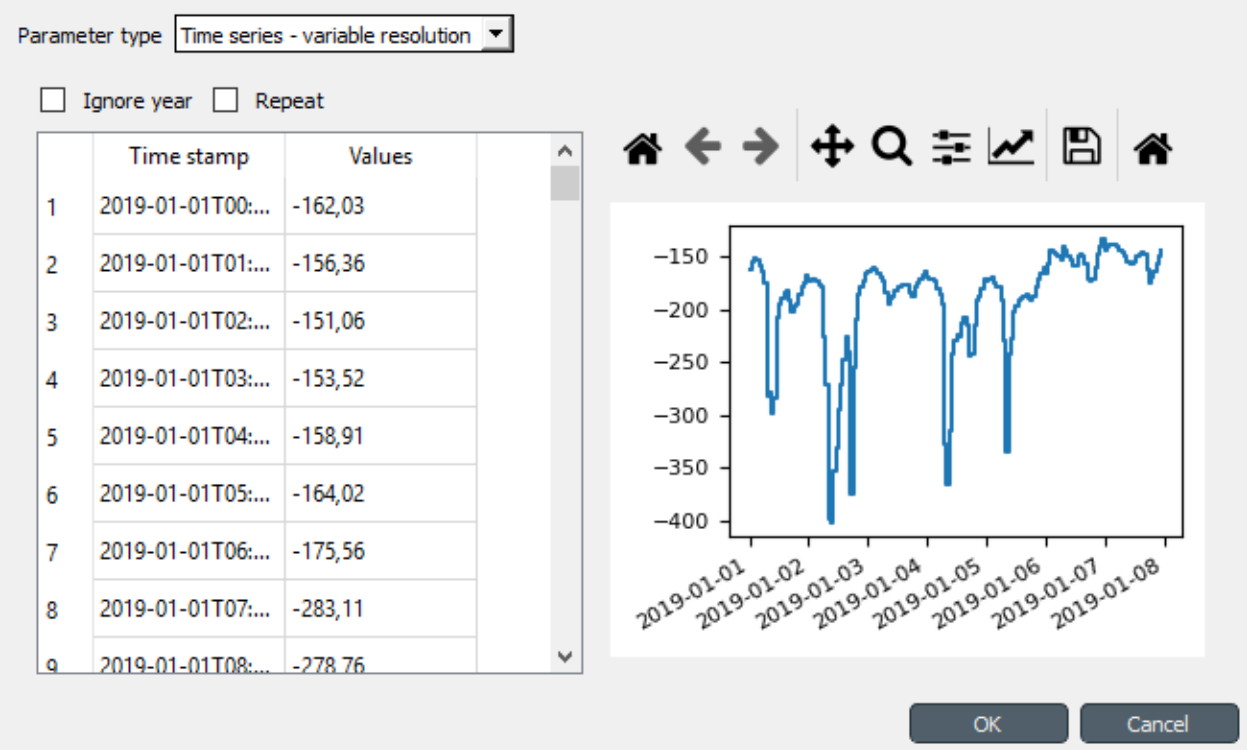
10.2 Plain values

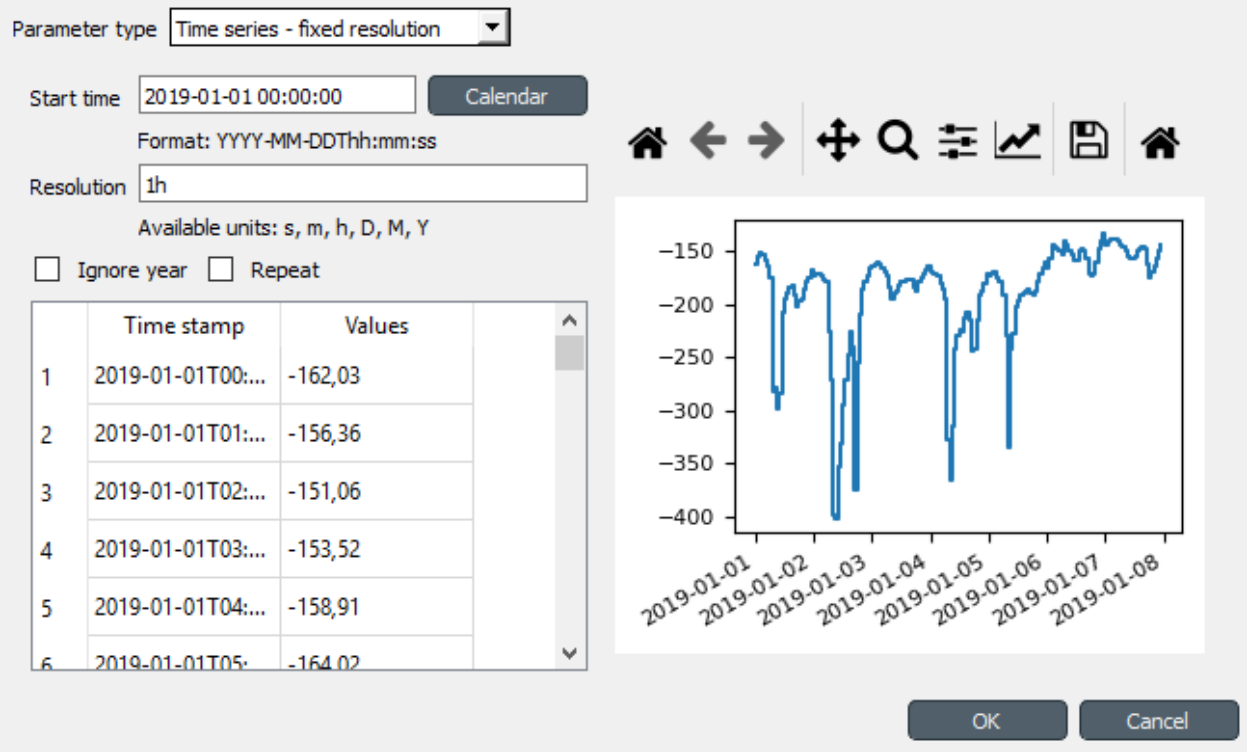
The simplest parameter values are of the *Plain value* type. These are numbers or booleans which can be set by entering `true` or `false` on the *Parameter value* field.



10.3 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 windows 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**. Data can be copied and pasted by **Ctrl-C** and **Ctrl-V**. Copying from/to an external spreadsheet program is supported.

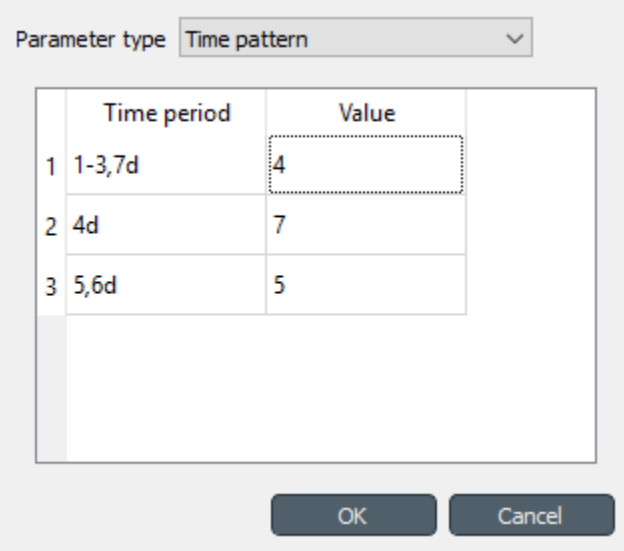
The time steps of a fixed resolution series are edited by the *Start time* and *Resolution* fields. The format for the start time is ISO8601. The *Resolution* field takes a single time step or a comma separated list of steps. If a list of resolution steps are provided then the steps are repeated so as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

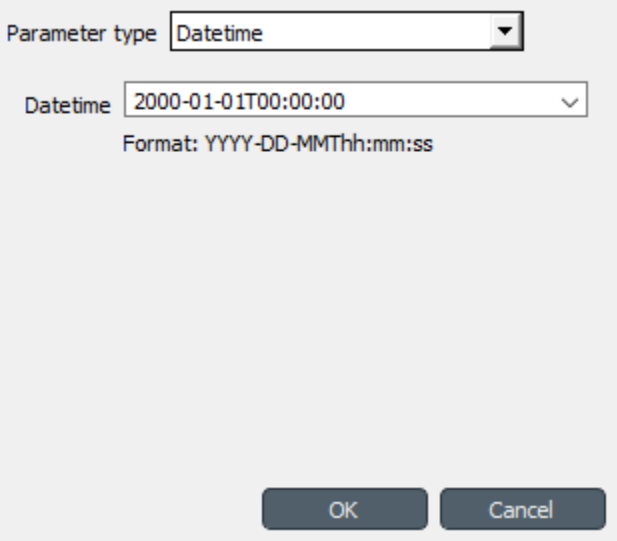
10.4 Time patterns

The time pattern editor holds a single table which shows the 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.



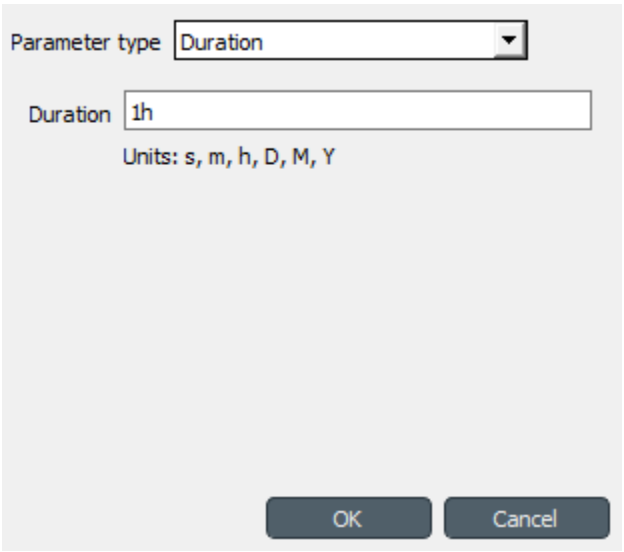
10.5 Datetimes

The datetime value should be entered in ISO8601 format.



10.6 Durations

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


Parameter type
Duration
Duration
1h
Units: s, m, h, D, M, Y
OK
Cancel

CHAPTER 11

Importing and exporting data

Note: This section is a work in progress.

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

11.1 Excel

In this section the excel import/export functionality is explained.

To import/export an excel file, select a **Data store** and open the **Tree view**. Then select **File -> Import** or **File -> Export** from the main menu.

11.1.1 Format

The excel files for import/export are formatted in the following way:

Tip: An easy way to get a excel template is to export an existing spine-database to excel.

Object classes:

	A	B	C
1	Sheet type	Data type	object class name
2	object	Parameter	unit
3			
4	**commodity_group**	**Parameter_1**	**Parameter_2**
5	named_unit_1		1,618
6	named_unit_2	2,718	3,14

Object timeseries:

	A	B	C
1	Sheet type	Data type	object class name
2	object	json array	unit
3			
4	node	named_unit_1	named_unit_2
5	json parameter	timseries_parameter	timseries_parameter
6	2018-01-01 00:00	1	3
7	2018-01-01 01:00	2	4
8	2018-01-01 02:00	3	

Relationship classes:

	A	B	C	D	E
1	Sheet type	Data type	relationship class name	Number of relationship dimensions	Number of pivoted relationship dimensions
2	relationship	Parameter	commodity_group__commodity	2	0
3					
4	commodity_group	commodity	relationship_parameter1	relationship_parameter2	
5	electricity_group	electricity		1	
6	water_group	water	1	2	
7					

Relationship timeseries:

	A	B	C	D	E	F	G
1	Sheet type	Data type	relationship class name	Number of relationship dimensions			
2	relationship	json array	unit__commodity	2			
3							
4	unit	SE1_spot	SE2_spot				
5	commodity	electricity	electricity				
6	json parameter	conversion_cost	conversion_cost				
7	2018-01-01 00:00	-24,03	-24,03				
8	2018-01-01 01:00	-24,03	-24,03				
9	2018-01-01 02:00	-24,02	-24,02				

When importing, all sheets with a valid format are imported, whereas sheets with invalid format are simply ignored. When exporting all object classes and relationship classes are exported. Only parameter values with timeseries data are exported in the timeseries format.

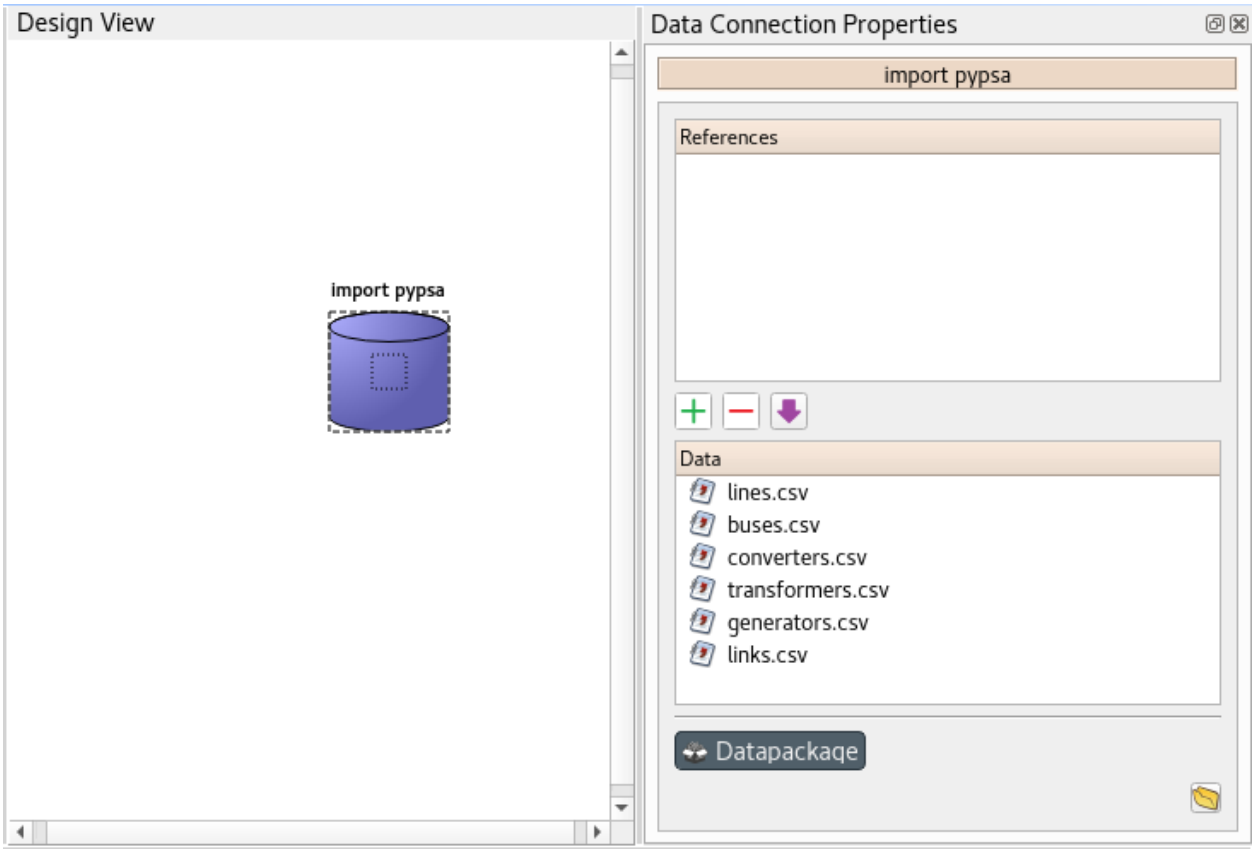
CHAPTER 12

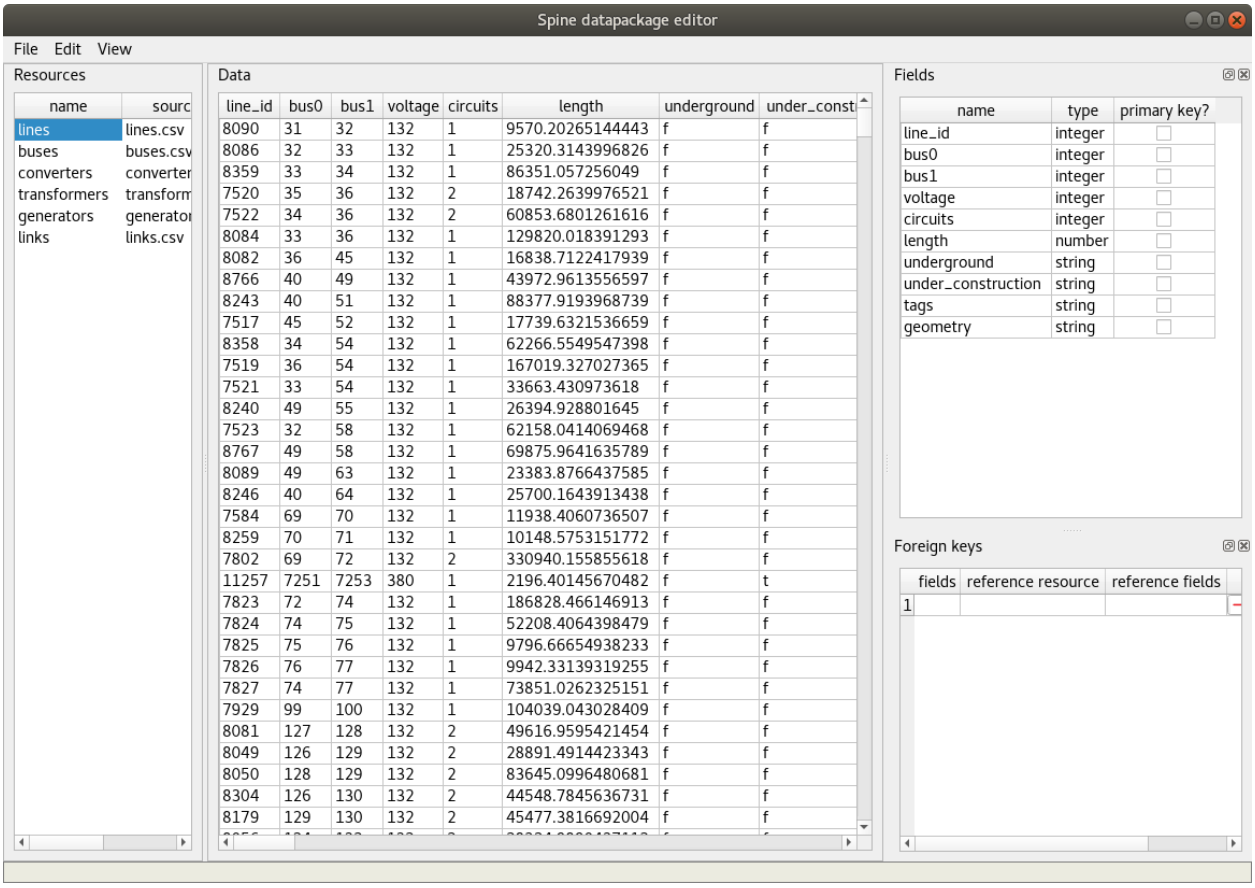
Spine datapackage editor

Note: This section is a work in progress.

This section describes the Spine datapackage editor, used to interact with tabular data and export it into Spine format.

To open the Spine datapackage editor, select a **Data Connection** with *CSV files* in it, and press the **Datapackage** button in its *Properties*:


Design View
import pypsa
Data Connection Properties
import pypsa
References
Data
lines.csv
buses.csv
converters.csv
transformers.csv
generators.csv
links.csv
Datapackage


Spine datapackage editor
File Edit View
Resources
name sourc
lines lines.csv
buses buses.csv
converters converter
transformers transform
generators generator
links links.csv
Data
line_id bus0 bus1 voltage circuits length underground under_const
8090 31 32 132 1 9570.20265144443 f f
8086 32 33 132 1 25320.3143996826 f f
8359 33 34 132 1 86351.057256049 f f
7520 35 36 132 2 18742.2639976521 f f
7522 34 36 132 2 60853.6801261616 f f
8084 33 36 132 1 129820.018391293 f f
8082 36 45 132 1 16838.7122417939 f f
8766 40 49 132 1 43972.9613556597 f f
8243 40 51 132 1 88377.9193968739 f f
7517 45 52 132 1 17739.6321536659 f f
8358 34 54 132 1 62266.5549547398 f f
7519 36 54 132 1 167019.327027365 f f
7521 33 54 132 1 33663.430973618 f f
8240 49 55 132 1 26394.928801645 f f
7523 32 58 132 1 62158.0414069468 f f
8767 49 58 132 1 69875.9641635789 f f
8089 49 63 132 1 23383.8766437585 f f
8246 40 64 132 1 25700.1643913438 f f
7584 69 70 132 1 11938.4060736507 f f
8259 70 71 132 1 10148.5753151772 f f
7802 69 72 132 2 330940.155855618 f f
11257 7251 7253 380 1 2196.40145670482 f t
7823 72 74 132 1 186828.466146913 f f
7824 74 75 132 1 52208.4064398479 f f
7825 75 76 132 1 9796.66654938233 f f
7826 76 77 132 1 9942.33139319255 f f
7827 74 77 132 1 73851.0262325151 f f
7929 99 100 132 1 104039.043028409 f f
8081 127 128 132 2 49616.9595421454 f f
8049 126 129 132 2 28891.4914423343 f f
8050 128 129 132 2 83645.0996480681 f f
8304 126 130 132 2 44548.7845636731 f f
8179 129 130 132 2 45477.3816692004 f f
Fields
name type primary key?
line_id integer
bus0 integer
bus1 integer
voltage integer
circuits integer
length number
underground string
under_construction string
tags string
geometry string
Foreign keys
fields reference resource reference fields
1

CHAPTER 13

Terminology

Here is a list of definitions that are used throughout the User Guide and in Spine Toolbox.

13.1 Spine Toolbox Terminology

- **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 (Data Interface) between the raw data and the Spine format database (Data Store).
- **Data Interface** 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.
- **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.
- **Project** is a Spine Toolbox concept and consists of a data processing chain that is built by the user for solving a particular problem. Current items that constitute a project are; Data Connection, Data Store, Tool, View, and a Data Interface. There can be any number of these items in a project, and they can be connected by drawing links between them.
- **Source directory** When in context of Tool templates, a Source directory is the directory where the main program file of the Tool template is located. This is also the recommended place where the Tool template definition file (.json) is saved.
- **Tool** is a project item that is used to execute Tool templates. To execute a script or a simulation model in Spine Toolbox, you attach a Tool template to a Tool.
- **Tool template** can be a computational process or a simulation model, or it can also be a script to convert data or calculate a new variable. Tool template takes some data as input and produces an output. Tool template contains a reference to the model code, external program that executes the code, and input data that the model code requires. Spine Model is a Tool template from Spine Toolbox's point-of-view.
- **View** A project item that can be used for visualizing project data.

- **Work directory** A directory where Tool template execution takes place. When a Tool is executed, Spine Toolbox creates a new *work* directory, copies all required and optional files needed for running the Tool template to this directory and executes it there. After execution has finished, output or result files can be archived into a timestamped results directory from the work directory.

13.2 Spine project Terminology

- **Case study** Spine project has 13 case studies that help to improve, validate and deploy different aspects of the Spine Model and Spine Toolbox.
- **Data Interface** is a component in Spine Toolbox, which handles connecting to and importing from external data sources.
- **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.
- **Scenario** A scenario combines data connections to form 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 Spine Model will be able to directly utilize as well as output them.
- **Spine Model** An interpreter, which formulates a solver-ready mixed-integer optimization problem based on the input data and the equations defined in the Spine Model. Outputs the solver results.
- **Use case** Potential way to use Spine Toolbox. Use cases together are used to test the functionality and stability of Spine Toolbox and Spine Model under different potential circumstances.

CHAPTER 14

Dependencies

Spine Toolbox requires Python 3.6 or higher.

Spine Toolbox uses code from packages and/or projects listed in the table below. Required packages must be installed for the application to start. Users can choose the SQL dialect API (pymysql, pyodbc psycopg2, and cx_Oracle) they want to use. These can be installed in Spine Toolbox when needed. If you want to deploy the application by using the provided *setup.py* file, you need to install *cx_Freeze* package (6.0b1 version or newer is recommended). All version numbers are minimum versions except for pyside2, where the version should be less than 5.12, which is not supported (yet).

14.1 Required packages

The following packages are available from `requirements.txt`

Package name	Version	License
pyside2	<5.12	LGPL
datapackage	1.2.3	MIT
qtconsole	4.3.1	BSD
sqlalchemy	1.2.6	MIT
openpyxl	2.5.0	MIT/Expat
spinedb_api	0.0.36	LGPL
numpy	1.15.1	BSD
matplotlib	3.0	BSD
scipy	1.1.0	BSD
jupyter-client	5.2.4	BSD
networkx	2.2	BSD
pymysql	0.9.2	MIT
pyodbc	4.0.23	MIT
psycopg2	2.7.4	LGPL
cx_Oracle	6.3.1	BSD
python-dateutil	2.8.0	PSF
pandas	0.24.0	BSD

14.2 Developer packages

The developer packages are available from `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	Version	License
black	19.3b0	MIT
pre-commit	1.16.1	MIT
pylint	2.3.0	GPL
sphinx	1.7.5	BSD
sphinx_rtd_theme	0.4.0	MIT
recommonmark	0.5.0	MIT
sphinx-autoapi	1.1.0	MIT

CHAPTER 15

Contribution Guide for Spine Toolbox

All are welcome to contribute!

15.1 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 if there's a sound 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.
- Other deviations from PEP-8 can be discussed if there are good reasons.

15.2 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. You can find a brief introduction to reStructured text in (http://www.sphinx-doc.org/en/stable/rest.html). 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.sh` on Linux to build the HTML pages. The created pages are found in `docs/build/html` directory.

15.3 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.sh` (Linux) 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. Avoid using style sheets in Qt Designer.

15.4 Reporting Bugs

This section is based on a set of best practices for open source projects (http://contribution-guide-org.readthedocs.io/)

15.4.1 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.

15.4.2 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.

15.5 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 issue, just join the conversation.

15.6 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 these instructions.

15.6.1 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 *dev* 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.

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.

15.6.2 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.

15.6.3 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 changelog entry crediting yourself.
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.

CHAPTER 16

API Reference

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

16.1 graphics_items

Classes for drawing graphics items on QGraphicsScene.

authors

M. Marin (KTH), P. Savolainen (VTT)

date 4.4.2018

16.1.1 Module Contents

class graphics_items.**ConnectorButton**(*parent*, *toolbox*, *position='left'*)

Bases: PySide2.QtWidgets.QGraphicsRectItem

Connector button graphics item. Used for Link drawing between project items.

parent

Project item bg rectangle

Type QGraphicsItem

toolbox

QMainWindow instance

Type ToolBoxUI

position

Either "top", "left", "bottom", or "right"

Type str

[1] Created with sphinx-autoapi

parent_name(*self*)
Returns project item name owning this connector button.

mousePressEvent(*self*, *event*)
Connector button mouse press event. Starts drawing a link.

Parameters **event** (*QGraphicsSceneMouseEvent*) – Event

mouseDoubleClickEvent(*self*, *event*)
Connector button mouse double click event. Makes sure the LinkDrawer is hidden.

Parameters **event** (*QGraphicsSceneMouseEvent*) – Event

hoverEnterEvent(*self*, *event*)
Sets a darker shade to connector button when mouse enters its boundaries.

Parameters **event** (*QGraphicsSceneMouseEvent*) – Event

hoverLeaveEvent(*self*, *event*)
Restore original brush when mouse leaves connector button boundaries.

Parameters **event** (*QGraphicsSceneMouseEvent*) – Event

class graphics_items.**ProjectItemIcon**(*toolbox*, *x*, *y*, *w*, *h*, *name*)
Bases: PySide2.QtWidgets.QGraphicsRectItem

Base class for Tool and View project item icons drawn in Design View.

toolbox
QMainWindow instance

Type ToolBoxUI

x
Icon x coordinate

Type int

y
Icon y coordinate

Type int

w
Icon width

Type int

h
Icon height

Type int

name
Item name

Type str

setup(*self*, *pen*, *brush*, *svg*, *svg_color*)
Setup item's attributes according to project item type. Intended to be called in the constructor's of classes that inherit from ItemImage class.

Parameters

- **pen** (*QPen*) – Used in drawing the background rectangle outline
- **brush** (*QBrush*) – Used in filling the background rectangle

- **svg** (*str*) – Path to SVG icon file
- **svg_color** (*QColor*) – Color of SVG icon

name(*self*)
Returns name of the item that is represented by this icon.

update_name_item(*self*, *new_name*)
Set a new text to name item. Used when a project item is renamed.

set_name_attributes(*self*)
Set name QGraphicsSimpleTextItem attributes (font, size, position, etc.)

conn_button(*self*, *position='left'*)
Returns items connector button (QWidget).

hoverEnterEvent(*self*, *event*)
Sets a drop shadow effect to icon when mouse enters its boundaries.

Parameters **event** (*QGraphicsSceneMouseEvent*) – Event

hoverLeaveEvent(*self*, *event*)
Disables the drop shadow when mouse leaves icon boundaries.

Parameters **event** (*QGraphicsSceneMouseEvent*) – Event

mouseMoveEvent(*self*, *event*)
Moves icon(s) while the mouse button is pressed. Update links that are connected to selected icons.

Parameters **event** (*QGraphicsSceneMouseEvent*) – Event

contextMenuEvent(*self*, *event*)
Show item context menu.

Parameters **event** (*QGraphicsSceneMouseEvent*) – Mouse event

keyPressEvent(*self*, *event*)
Handles deleting and rotating the selected item when dedicated keys are pressed.

Parameters **event** (*QKeyEvent*) – Key event

itemChange(*self*, *change*, *value*)
Destroys the drop shadow effect when the items is removed from a 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

show_item_info(*self*)
Update GUI to show the details of the selected item.

class graphics_items.**DataConnectionIcon**(*toolbox*, *x*, *y*, *w*, *h*, *name*)
Bases: *graphics_items.ProjectItemIcon*

Data Connection icon for the Design View.

toolbox
QMainWindow instance

Type ToolBoxUI

x
Icon x coordinate

Type int

y
Icon y coordinate

Type int

w
Width of master icon

Type int

h
Height of master icon

Type int

name
Item name

Type str

dragEnterEvent(*self*, *event*)
Drag and drop action enters. Accept file drops from the filesystem.

Parameters **event** (*QGraphicsSceneDragDropEvent*) – Event

dragLeaveEvent(*self*, *event*)
Drag and drop action leaves.

Parameters **event** (*QGraphicsSceneDragDropEvent*) – Event

dragMoveEvent(*self*, *event*)
Accept event.

dropEvent(*self*, *event*)
Emit files_dropped_on_dc signal from scene, with this instance, and a list of files for each dropped url.

select_on_drag_over(*self*)
Called when the timer started in drag_enter_event is elapsed. Select this item if the drag action is still over it.

class graphics_items.**ToolIcon**(*toolbox*, *x*, *y*, *w*, *h*, *name*)
Bases: *graphics_items.ProjectItemIcon*

Tool image with a rectangular background, an SVG icon, a name label, and a connector button.

toolbox
QMainWindow instance

Type ToolBoxUI

x
Icon x coordinate

Type int

y
Icon y coordinate

Type int

w
Width of master icon

Type int

h
Height of master icon

Type int

name
Item name

Type str

value_for_time(*self*, *msecs*)

start_animation(*self*)
Start the animation that plays when the Tool associated to this GraphicsItem is running.

stop_animation(*self*)
Stop animation

class graphics_items.**DataStoreIcon**(*toolbox*, *x*, *y*, *w*, *h*, *name*)
Bases: *graphics_items.ProjectItemIcon*

Data Store item that is drawn into QGraphicsScene. NOTE: Make sure to set self._master as the parent of all drawn items. This groups the individual QGraphicsItems together.

toolbox
QMainWindow instance

Type ToolBoxUI

x
Icon x coordinate

Type int

y
Icon y coordinate

Type int

w
Width of master icon

Type int

h
Height of master icon

Type int

name
Item name

Type str

class graphics_items.**ViewIcon**(*toolbox*, *x*, *y*, *w*, *h*, *name*)
Bases: *graphics_items.ProjectItemIcon*

View icon for the Design View

toolbox
QMainWindow instance

Type ToolBoxUI

x
Icon x coordinate

Type int

y
Icon y coordinate

Type int

w
Width of background rectangle

Type int

h
Height of background rectangle

Type int

name
Item name

Type str

class graphics_items.**DataInterfaceIcon**(*toolbox*, *x*, *y*, *w*, *h*, *name*)
Bases: *graphics_items.ProjectItemIcon*

Data Interface item that is drawn into QGraphicsScene. NOTE: Make sure to set self._master as the parent of all drawn items. This groups the individual QGraphicsItems together.

toolbox
QMainWindow instance

Type ToolBoxUI

x
Icon x coordinate

Type int

y
Icon y coordinate

Type int

w
Width of master icon

Type int

h
Height of master icon

Type int

name
Item name

Type str

class graphics_items.**Link**(*toolbox*, *src_connector*, *dst_connector*)
Bases: PySide2.QtWidgets.QGraphicsPathItem

An item that represents a connection between project items.

toolbox
main UI class instance

Type *ToolboxUI*

src_connector
Source connector button

Type *ConnectorButton*

dst_connector
Destination connector button

Type *ConnectorButton*

find_model_index(*self*)
Find model index from connection model.

find_parallel_link(*self*)
Find parallel link.

send_to_bottom(*self*)
Send link behind other links.

mousePressEvent(*self*, *e*)
Trigger slot button if it is underneath.

Parameters **e** (*QGraphicsSceneMouseEvent*) – Mouse event

mouseDoubleClickEvent(*self*, *e*)
Accept event to prevent unwanted feedback links to be created when propagating this event to connector buttons underneath.

contextMenuEvent(*self*, *e*)
Show context menu unless mouse is over one of the slot buttons.

Parameters **e** (*QGraphicsSceneMouseEvent*) – Mouse event

keyPressEvent(*self*, *event*)
Remove associated connection if this is selected and delete is pressed.

update_geometry(*self*)
Update path.

paint(*self*, *painter*, *option*, *widget*)
Set pen according to selection state.

itemChange(*self*, *change*, *value*)
Bring selected link to top.

class graphics_items.**LinkDrawer**
Bases: PySide2.QtWidgets.QGraphicsPathItem

An item that allows one to draw links between slot buttons in QGraphicsView.

start_drawing_at(*self*, *src_rect*)
Start drawing from the center point of the clicked button.

Parameters **src_rect** (*QRecF*) – Rectangle of the clicked button

update_geometry(*self*)
Update path.

class graphics_items.**ObjectItem**(*graph_view_form*, *object_name*, *object_class_id*, *object_class_name*, *x*, *y*, *extent*, *object_id=0*, *label_color=Qt.transparent*)
Bases: PySide2.QtWidgets.QGraphicsPixmapItem

Object item to use with GraphViewForm.

graph_view_form
'owner'

Type *GraphViewForm*

object_name
object name

Type str

object_class_id
object class id

Type int

object_class_name
object class name

Type str

x
x-coordinate of central point

Type float

y
y-coordinate of central point

Type float

extent
preferred extent

Type int

object_id
object id (for filtering parameters)

Type int

label_font
label font

Type QFont

label_color
label bg color

Type QColor

shape(*self*)
Make the entire bounding rect to be the shape.

paint(*self*, *painter*, *option*, *widget=None*)
Try and make it more clear when an item is selected.

make_template(*self*)
Make this object par of a template for a relationship.

remove_template(*self*)
Make this arc no longer a template.

edit_name(*self*)
Start editing object name.

finish_name_editing(*self*)
Called by the label item when editing finishes.

add_incoming_arc_item(*self*, *arc_item*)
Add an ArcItem to the list of incoming arcs.

add_outgoing_arc_item(*self*, *arc_item*)
Add an ArcItem to the list of outgoing arcs.

keyPressEvent(*self*, *event*)
Triggers name editing.

mouseDoubleClickEvent(*self*, *event*)
Triggers name editing.

mousePressEvent(*self*, *event*)
Saves original position.

mouseMoveEvent(*self*, *event*)
Calls move related items and checks for a merge target.

mouseReleaseEvent(*self*, *event*)
Merge, bounce, or just do nothing.

check_for_merge_target(*self*, *scene_pos*)
Checks if this item is touching another item so they can merge (this happens when building a relationship).

merge_item(*self*, *other*)
Merges this item with another. Tries to create a relationship if needed.

add_into_relationship(*self*)
Try and add this item into a relationship between the buddies.

move_related_items_by(*self*, *pos_diff*)
Moves related items.

contextMenuEvent(*self*, *e*)
Shows context menu.

Parameters **e** (*QGraphicsSceneMouseEvent*) – Mouse event

set_all_visible(*self*, *on*)
Sets visibility status for this item and all related items.

wipe_out(*self*)
Removes this item and all related items from the scene.

class graphics_items.**ArcItem**(*graph_view_form*, *relationship_class_id*, *src_item*, *dst_item*, *width*, *arc_color*, *object_id_list=''*, *token_color=QColor()*, *token_object_extent=0*, *token_object_label_color=QColor()*, *token_object_name_tuple_list=()*)
Bases: PySide2.QtWidgets.QGraphicsLineItem

Arc item to use with GraphViewForm.

graph_view_form
'owner'

Type *GraphViewForm*

relationship_class_id
relationship class id

Type int

src_item
source item

Type *ObjectItem*

dst_item
destination item

Type *ObjectItem*

width
Preferred line width

Type int

arc_color
arc color

Type QColor

object_id_list
object id comma separated list

Type str

token_object_extent
token preferred extent

Type int

token_color
token bg color

Type QColor

token_object_name_tuple_list
token (object class name, object name) tuple list

Type list

paint(*self*, *painter*, *option*, *widget=None*)
Try and make it more clear when an item is selected.

make_template(*self*)
Make this arc part of a template for a relationship.

remove_template(*self*)
Make this arc no longer part of a template for a relationship.

move_src_by(*self*, *pos_diff*)
Move source point by pos_diff. Used when moving ObjectItems around.

move_dst_by(*self*, *pos_diff*)
Move destination point by pos_diff. Used when moving ObjectItems around.

hoverEnterEvent(*self*, *event*)
Set viewport's cursor to arrow.

hoverLeaveEvent(*self*, *event*)
Restore viewport's cursor.

class graphics_items.**ObjectLabelItem**(*object_item*, *text*, *width*, *bg_color*)
Bases: PySide2.QtWidgets.QGraphicsTextItem

Object label item to use with GraphViewForm.

object_item
the ObjectItem instance

Type *ObjectItem*

text
text

Type str

width
maximum width

Type int

bg_color
color to paint the label

Type QColor

set_bg_color(*self*, *bg_color*)
Set background color.

set_full_text(*self*)

set_text(*self*, *text*)
Store real text, and then try and fit it as best as possible in the width (reduce font point size, elide text...)

keyPressEvent(*self*, *event*)
Give up focus when the user presses Enter or Return. In the meantime, adapt item geometry so text is always centered.

focusOutEvent(*self*, *event*)
Call method to finish name editing in object item.

class graphics_items.**ArcTokenItem**(*arc_item*, *color*, *object_extent*, *object_label_color*, **object_name_tuples*)
Bases: PySide2.QtWidgets.QGraphicsEllipseItem

Arc token item to use with GraphViewForm.

arc_item
the ArcItem instance

Type *ArcItem*

color
color to paint the token

Type QColor

object_extent
Preferred extent

Type int

object_label_color
Preferred extent

Type QColor

object_name_tuples
one or more (object class name, object name) tuples

Type Iterable

update_pos(*self*)
Put token item in position.

class graphics_items.**SimpleObjectItem**(*parent*, *extent*, *label_color*, *object_class_name*, *object_name*)
Bases: PySide2.QtWidgets.QGraphicsPixmapItem

Object item to use with GraphViewForm.

parent
arc token item

Type *ArcTokenItem*

extent
preferred extent

Type int

label_color
label bg color

Type QColor

object_class_name
object class name

Type str

object_name
object name

Type str

setOffset(*self*, *offset*)

class graphics_items.**OutlinedTextItem**(*text*, *font*, *brush=QBrush(Qt.black)*, *outline_pen=QPen(Qt.white*, *3*, *Qt.SolidLine)*)
Bases: PySide2.QtWidgets.QGraphicsSimpleTextItem

Outlined text item to use with GraphViewForm.

text
text to show

Type str

font
font to display the text

Type QFont

brush

Type QBrus

outline_pen

Type QPen

class graphics_items.**CustomTextItem**(*html*, *font*)
Bases: PySide2.QtWidgets.QGraphicsTextItem

Custom text item to use with GraphViewForm.

html
text to show

Type str

font
font to display the text

Type QFont

16.2 project

Spine Toolbox project class.

authors

P. Savolainen (VTT), E. Rinne (VTT)

date 10.1.2018

16.2.1 Module Contents

class project.**SpineToolboxProject**(*toolbox*, *name*, *description*, *work_dir=None*, *ext='.proj'*)
Bases: *metaobject.MetaObject*

Class for Spine Toolbox projects.

toolbox
toolbox of this project

Type *ToolboxUI*

name
Project name

Type str

description
Project description

Type str

work_dir
Project work directory

Type str

ext
Project save file extension(.proj)

Type str

change_name(*self*, *name*)
Changes project name and updates project dir and save file name.

Parameters **name** (*str*) – Project (long) name

change_filename(*self*, *new_filename*)
Change the save filename associated with this project.

Parameters **new_filename** (*str*) – Filename used in saving the project. No full path. Example 'project.proj'

change_work_dir(*self*, *new_work_path*)
Change project work directory.

Parameters **new_work_path** (*str*) – Absolute path to new work directory

rename_project(*self*, *name*)
Save project under a new name. Used with File->Save As... menu command. Checks if given project name is valid.

Parameters **name** (*str*) – New (long) name for project

save(*self*, *tool_def_paths*)
Collect project information and objects into a dictionary and write to a JSON file.

Parameters **tool_def_paths** (*list*) – List of paths to tool definition files

load(*self*, *item_dict*)
Populate project item model with items loaded from project file.

Parameters **item_dict** (*dict*) – Dictionary containing all project items in JSON format

Returns Boolean value depending on operation success.

load_tool_template_from_file(*self*, *jsonfile*)
Create a Tool template according to a tool definition file.

Parameters **jsonfile** (*str*) – Path of the tool template definition file

Returns Instance of a subclass if Tool

load_tool_template_from_dict(*self*, *definition*, *path*)
Create a Tool template according to a dictionary.

Parameters

- **definition** (*dict*) – Dictionary with the tool definition
- **path** (*str*) – Folder of the main program file

Returns Instance of a subclass if Tool

add_data_store(*self*, *name*, *description*, *url*, *x=0*, *y=0*, *set_selected=False*, *verbosity=True*)
Adds a Data Store to project item model.

Parameters

- **name** (*str*) – Name
- **description** (*str*) – Description of item
- **url** (*dict*) – Url information
- **x** (*int*) – X coordinate of item on scene
- **y** (*int*) – Y coordinate of item on scene
- **set_selected** (*bool*) – Whether to set item selected after the item has been added to project
- **verbosity** (*bool*) – If True, prints message

add_data_connection(*self*, *name*, *description*, *references*, *x=0*, *y=0*, *set_selected=False*, *verbosity=True*)
Adds a Data Connection to project item model.

Parameters

- **name** (*str*) – Name
- **description** (*str*) – Description of item
- **references** (*list (str)*) – List of file paths

- **x** (*int*) – X coordinate of item on scene
- **y** (*int*) – Y coordinate of item on scene
- **set_selected** (*bool*) – Whether to set item selected after the item has been added to project
- **verbosity** (*bool*) – If True, prints message

add_tool(*self*, *name*, *description*, *tool_template*, *use_work=True*, *x=0*, *y=0*, *set_selected=False*, *verbosity=True*)

Adds a Tool to project item model.

Parameters

- **name** (*str*) – Name
- **description** (*str*) – Description of item
- **tool_template** (ToolTemplate) – Tool template of this tool
- **use_work** (*bool*) – Execute in work directory
- **x** (*int*) – X coordinate of item on scene
- **y** (*int*) – Y coordinate of item on scene
- **set_selected** (*bool*) – Whether to set item selected after the item has been added to project
- **verbosity** (*bool*) – If True, prints message

add_view(*self*, *name*, *description*, *x=0*, *y=0*, *set_selected=False*, *verbosity=True*)

Adds a View to project item model.

Parameters

- **name** (*str*) – Name
- **description** (*str*) – Description of item
- **x** (*int*) – X coordinate of item on scene
- **y** (*int*) – Y coordinate of item on scene
- **set_selected** (*bool*) – Whether to set item selected after the item has been added to project
- **verbosity** (*bool*) – If True, prints message

add_data_interface(*self*, *name*, *description*, *import_file_path=''*, *mappings=None*, *x=0*, *y=0*, *set_selected=False*, *verbosity=True*)

Adds a Data Interface to project item model.

Parameters

- **name** (*str*) – Name
- **description** (*str*) – Description of item
- **x** (*int*) – X coordinate of item on scene
- **y** (*int*) – Y coordinate of item on scene
- **set_selected** (*bool*) – Whether to set item selected after the item has been added to project
- **verbosity** (*bool*) – If True, prints message

append_connection_model(*self*, *item_name*, *category*)
Adds new item to connection model to keep project and connection model synchronized.

add_to_dag(*self*, *item_name*)
Add new directed graph object.

set_item_selected(*self*, *item*)
Sets item selected and shows its info screen.

Parameters **item** (ProjectItem) – Project item to select

execute_selected(*self*)
Starts executing selected directed acyclic graph. Selected graph is determined by the selected project item(s). Aborts, if items from multiple graphs are selected.

execute_project(*self*)
Determines the number of directed acyclic graphs to execute in the project. Determines the execution order of project items in each graph. Creates an instance for executing the first graph and starts executing it.

graph_execution_finished(*self*, *state*)
Releases resources from previous execution and prepares the next graph for execution if there are still graphs left. Otherwise, finishes the run.

Parameters **state** (*int*) – 0: Ended normally. -1: User pressed Stop button

stop(*self*)
Stops execution of the current DAG. Slot for the main window Stop tool button in the toolbar.

handle_invalid_graphs(*self*)
Prints messages to Event Log if there are invalid DAGs (e.g. contain self-loops) in the project.

export_graphs(*self*)
Export all valid directed acyclic graphs in project to GraphML files.

16.3 parameter_value_formatting

Functions for textual display of parameter values in table views.

authors

A. Soininen (VTT)

date 12.7.2019

16.3.1 Module Contents

parameter_value_formatting.**format_for_DisplayRole**(*value_in_database*)
Returns the value's database representation formatted for Qt.DisplayRole.

parameter_value_formatting.**format_for_EditRole**(*value_in_database*)
Returns the value's database representation formatted for Qt.EditRole.

parameter_value_formatting.**format_for_ToolTipRole**(*value_in_database*)
Returns the value's database representation formatted for Qt.ToolTipRole.

16.4 config

Application constants and style sheets

author

P. Savolainen (VTT)

date 2.1.2018

16.4.1 Module Contents

config.**SPINE_TOOLBOX_VERSION = 0.3**

config.**REQUIRED_SPINEDB_API_VERSION = 0.0.36**

config.**INVALID_CHARS = ['<', '>', ':', '"', '/', '\\', '|', '?', '*', '.']**

config.**INVALID_FILENAME_CHARS = ['<', '>', ':', '"', '/', '\\', '|', '?', '*']**

config.**APPLICATION_PATH**

config.**TOOL_OUTPUT_DIR = output**

config.**GAMS_EXECUTABLE = gams**

config.**JULIA_EXECUTABLE = julia**

config.**PYTHON_EXECUTABLE = python3**

config.**TOOL_TYPES = ['Julia', 'Python', 'GAMS', 'Executable']**

config.**REQUIRED_KEYS = ['name', 'tooltype', 'includes']**

config.**OPTIONAL_KEYS = ['description', 'short_name', 'inputfiles', 'inputfiles_opt', 'outpu**

config.**LIST_REQUIRED_KEYS = ['includes', 'inputfiles', 'inputfiles_opt', 'outputfiles']**

config.**JL_REPL_TIME_TO_DEAD = 5.0**

config.**JL_REPL_RESTART_LIMIT = 3**

config.**STATUSBAR_SS = QStatusBar{background-color: #EBEBE0;border-width: 1px;border-colo**

config.**SETTINGS_SS = #SettingsForm{background-color: ghostwhite;}QLabel{color: black;}QL**

config.**ICON_TOOLBAR_SS = QToolBar{spacing: 6px; background: qlineargradient(x1: 1, y1:**

config.**PARAMETER_TAG_TOOLBAR_SS**

config.**TEXTBROWSER_SS = QTextBrowser {background-color: #19232D; border: 1px solid #3241**

config.**MAINWINDOW_SS = QMainWindow::separator{width: 3px; background-color: lightgray; bo**

config.**TREEVIEW_HEADER_SS = QHeaderView::section{background-color: #ecd8c6; font-size: 1**

16.5 models

Classes for handling models in PySide2's model/view framework. Note: These are Spine Toolbox internal data models.

authors

P. Savolainen (VTT), M. Marin (KTH), P. Vennström (VTT)

date 23.1.2018

16.5.1 Module Contents

class models.**ProjectItemModel**(*toolbox*, *root*)
Bases: PySide2.QtCore.QAbstractItemModel

Class to store project items, e.g. Data Stores, Data Connections, Tools, Views.

toolbox
QMainWindow instance

Type *ToolboxUI*

root
Root item for the project item tree

Type *ProjectItem*

root(*self*)
Returns root project item.

rowCount(*self*, *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(*self*, *parent=QModelIndex()*)
Returns model column count.

flags(*self*, *index*)
Returns flags for the item at given index

Parameters **index** (*QModelIndex*) – Flags of item at this index.

parent(*self*, *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(*self*, *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(*self*, *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

project_item(*self*, *index*)
Returns project item at given index.

Parameters **index** (*QModelIndex*) – Index of project item

Returns Item at given index or root project item if index is not valid

Return type *ProjectItem*

find_category(*self*, *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(*self*, *name*)
Returns the QModelIndex of the project 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

insert_item(*self*, *item*, *parent=QModelIndex()*)
Adds a new item to model. Fails if given parent is not a category item nor a root item. New item is inserted as the last item.

Parameters

- **item** (ProjectItem) – Project item to add to model
- **parent** (*QModelIndex*) – Parent project item

Returns True if successful, False otherwise

Return type bool

remove_item(*self*, *item*, *parent=QModelIndex()*)
Removes item from model.

Parameters

- **item** (ProjectItem) – Project 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

setData(*self*, *index*, *value*, *role=Qt.EditRole*)
Changes the name of the project item at given index to given value. # TODO: If the item is a Data Store the reference sqlite path must be updated.

Parameters

- **index** (*QModelIndex*) – Project item index
- **value** (*str*) – New project item name
- **role** (*int*) – Item data role to set

Returns True or False depending on whether the new name is acceptable.

Return type bool

items(*self*, *category_name=None*)
Returns a list of items in model according to category name. If no category name given, returns all project items in a list.

Parameters **category_name** (*str*) – Item category. Data Connections, Data Stores, Tools or Views permitted.

Returns obj:'list' of :obj:'ProjectItem': 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(*self*)
Returns the number of all project items in the model excluding category items and root.

Returns Number of items

Return type int

item_names(*self*)
Returns all project item names in a list.

Returns 'list' of obj:'str': Item names

Return type obj

new_item_index(*self*, *category*)
Returns the index where a new item can be appended according to category. This is needed for appending the connection model.

Parameters **category** (*str*) – Display Role of the parent

Returns Number of items according to category

Return type int

short_name_reserved(*self*, *short_name*)
Checks if the directory name derived from the name of the given item is in use.

Parameters **short_name** (*str*) – Item short name

Returns True if short name is taken, False if it is available.

Return type bool

class models.**ToolTemplateModel**(*toolbox=None*)
Bases: PySide2.QtCore.QAbstractListModel

Class to store tools that are available in a project e.g. GAMS or Julia models.

rowCount(*self*, *parent=None*)
Must be reimplemented when subclassing. Returns the number of Tools in the model.

Parameters **parent** (*QModelIndex*) – Not used (because this is a list)

Returns Number of rows (available tools) in the model

data (*self*, *index*, *role=None*)
Must be reimplemented when subclassing.

Parameters

- **index** (*QModelIndex*) – Requested index
- **role** (*int*) – Data role

Returns Data according to requested role

flags (*self*, *index*)
Returns enabled flags for the given index.

Parameters **index** (*QModelIndex*) – Index of Tool

insertRow (*self*, *tool*, *row=None*, *parent=QModelIndex()*)
Insert row (tool) into model.

Parameters

- **tool** (*Tool*) – Tool added to the model
- **row** (*str*) – Row to insert tool to
- **parent** (*QModelIndex*) – Parent of child (not used)

Returns Void

removeRow (*self*, *row*, *parent=QModelIndex()*)
Remove row (tool) from model.

Parameters

- **row** (*int*) – Row to remove the tool from
- **parent** (*QModelIndex*) – Parent of tool on row (not used)

Returns Boolean variable

update_tool_template (*self*, *tool*, *row*)
Update tool template.

Parameters

- **tool** (*ToolTemplate*) – new tool, to replace the old one
- **row** (*int*) – Position of the tool to be updated

Returns Boolean value depending on the result of the operation

tool_template (*self*, *row*)
Returns tool template on given row.

Parameters **row** (*int*) – Row of tool template

Returns ToolTemplate from tool template list or None if given row is zero

find_tool_template (*self*, *name*)
Returns tool template with the given name.

Parameters **name** (*str*) – Name of tool template to find

tool_template_row (*self*, *name*)
Returns the row on which the given template is located or -1 if it is not found.

tool_template_index(*self*, *name*)
Returns the QModelIndex on which a tool template with the given name is located or invalid index if it is not found.

class models.**ConnectionModel**(*toolbox=None*)
Bases: PySide2.QtCore.QAbstractTableModel

Table model for storing connections between items.

flags(*self*, *index*)
Returns flags for table items.

rowCount(*self*, **args*, ***kwargs*)
Number of rows in the model. This should be the same as the number of items in the project.

columnCount(*self*, **args*, ***kwargs*)
Number of columns in the model. This should be the same as the number of items in the project.

headerData(*self*, *section*, *orientation*, *role=Qt.DisplayRole*)
Returns header data according to given role.

setHeaderData(*self*, *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(*self*, *index*, *role*)
Returns the data stored under the given role for the item referred to by the index. DisplayRole is a string "False" or "True" depending on if a Link is present.

Parameters

- **index** (*QModelIndex*) – Index of item
- **role** (*int*) – Data role

Returns Item data for given role.

setData(*self*, *index*, *value*, *role=Qt.EditRole*)
Set data of single cell in table. Toggles the checkbox state at index.

Parameters

- **index** (*QModelIndex*) – Index of data to edit
- **value** (*QVariant*) – Value to write to index (Link instance)
- **role** (*int*) – Role for editing

insertRows(*self*, *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(*self*, *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

_rowRemovalPossible(*self*, *row*, *count*)

removeRows(*self*, *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

_columnRemovalPossible(*self*, *column*, *count*)

removeColumns(*self*, *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

append_item(*self*, *name*, *index*)
Embiggens connections table by a new item.

Parameters

- **name** (*str*) – New item name
- **index** (*int*) – Table row and column where the new item is appended

Returns True if successful, False otherwise

remove_item(*self*, *name*)
Removes project item from connections table.

Parameters **name** (*str*) – Name of removed item

Returns True if successful, False otherwise

output_items(*self*, *name*)
Returns a list of output items for the given item.

Parameters **name** (*str*) – Project item name

Returns Output project item names in a list if they exist or an empty list if they don't.

Return type (list)

input_items(*self*, *name*)
Returns a list of input items for the given item.

Parameters **name** (*str*) – Project item name

Returns Input project item names in a list if they exist or an empty list if they don't.

Return type (list)

get_connections (*self*)
Returns the internal data structure of the model.

connected_links (*self*, *name*)
Returns a list of connected links for the given item

reset_model (*self*, *connection_table*)
Reset model. Used in replacing the current model with a boolean table that represents connections. Overwrites the current model with a True or False (boolean) table that is read from a project save file (.json). This table is updated by restore_links() method to add Link instances to True cells and Nones to False cells.

find_index_in_header (*self*, *name*)
Returns the row or column (row==column) of the header item with the given text (item name).

link (*self*, *row*, *column*)
Returns Link instance stored on row and column.

class models.**MinimalTableModel** (*parent=None*)
Bases: PySide2.QtCore.QAbstractTableModel

Table model for outlining simple tabular data.

parent
the parent widget, usually an instance of TreeViewForm

Type QMainWindow

clear (*self*)
Clear all data in model.

flags (*self*, *index*)
Return index flags.

rowCount (*self*, *parent=QModelIndex()*)
Number of rows in the model.

columnCount (*self*, *parent=QModelIndex()*)
Number of columns in the model.

headerData (*self*, *section*, *orientation=Qt.Horizontal*, *role=Qt.DisplayRole*)
Get headers.

set_horizontal_header_labels (*self*, *labels*)
Set horizontal header labels.

insert_horizontal_header_labels (*self*, *section*, *labels*)
Insert horizontal header labels at the given section.

horizontal_header_labels (*self*)

setHeaderData (*self*, *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 (*self*, *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(*self*, *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.

column_data(*self*, *column*, *role=Qt.DisplayRole*)
Returns the data stored under the given role for the given column.

Parameters

- **column** (*int*) – Item column
- **role** (*int*) – Data role

Returns Column data for given role.

model_data(*self*, *role=Qt.DisplayRole*)
Returns the data stored under the given role in the entire model.

Parameters **role** (*int*) – Data role

Returns Model data for given role.

setData(*self*, *index*, *value*, *role=Qt.EditRole*)
Set data in model.

batch_set_data(*self*, *indexes*, *data*)
Batch set data for indexes.

insertRows(*self*, *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(*self*, *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(*self*, *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(*self*, *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(*self*, *main_data=None*)
Reset model.

class models.**EmptyRowModel**(*parent=None*)
Bases: *models.MinimalTableModel*

A table model with a last empty row.

flags(*self*, *index*)
Return default flags except if forcing defaults.

set_default_row(*self*, ***kwargs*)
Set default row data.

clear(*self*)

reset_model(*self*, *data*)

_handle_data_changed(*self*, *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.

_handle_rows_removed(*self*, *parent*, *first*, *last*)
Insert a new empty row in case it's been removed.

_handle_rows_inserted(*self*, *parent*, *first*, *last*)
Handle rowsInserted signal.

set_rows_to_default(*self*, *first*, *last*)
Set default data in newly inserted rows.

class models.**HybridTableModel**(*parent=None*)
Bases: *models.MinimalTableModel*

A model that concatenates two models, one for existing items and another one for new items.

flags(*self*, *index*)
Return flags for given index. Depending on the index's row we will land on one of the two models.

data(*self*, *index*, *role=Qt.DisplayRole*)
Return data for given index and role. Depending on the index's row we will land on one of the two models.

rowCount (*self*, *parent=QModelIndex()*)
Return the sum of rows in the two models.

batch_set_data (*self*, *indexes*, *data*)
Batch set data for indexes. Distribute indexes and data among the two models and call batch_set_data on each of them.

insertRows (*self*, *row*, *count*, *parent=QModelIndex()*)
Find the right sub-model (or the empty model) and call insertRows on it.

removeRows (*self*, *row*, *count*, *parent=QModelIndex()*)
Find the right sub-models (or empty model) and call removeRows on them.

set_horizontal_header_labels (*self*, *labels*)

reset_model (*self*, *data*)
Reset model data.

_handle_new_item_model_rows_inserted (*self*, *parent*, *first*, *last*)

class models.**DatapackageResourcesModel** (*parent*)
Bases: *models.MinimalTableModel*

A model of datapackage resource data, used by SpineDatapackageWidget.

parent

Type *SpineDatapackageWidget*

reset_model (*self*, *resources*)

flags (*self*, *index*)

class models.**DatapackageFieldsModel** (*parent*)
Bases: *models.MinimalTableModel*

A model of datapackage field data, used by SpineDatapackageWidget.

parent

Type *SpineDatapackageWidget*

reset_model (*self*, *schema*)

class models.**DatapackageForeignKeysModel** (*parent*)
Bases: *models.EmptyRowModel*

A model of datapackage foreign key data, used by SpineDatapackageWidget.

parent

Type *SpineDatapackageWidget*

reset_model (*self*, *foreign_keys*)

class models.**TableModel** (*headers=None*, *data=None*)
Bases: PySide2.QtCore.QAbstractItemModel

Used by custom_qtableview.FrozenTableView

parent (*self*, *child=None*)

index (*self*, *row*, *column*, *parent=QModelIndex()*)

set_data (*self*, *data*, *headers*)

rowCount (*self*, *parent=QModelIndex()*)

columnCount (*self*, *parent=QModelIndex()*)

headerData (*self*, *section*, *orientation*, *role*)

row (*self*, *index*)

data (*self*, *index*, *role*)

16.6 project_item

ProjectItem class.

authors

P. Savolainen (VTT)

date 4.10.2018

16.6.1 Module Contents

class project_item.**ProjectItem**(*name*, *description*, *is_root=False*, *is_category=False*)

Bases: *metaobject.MetaObject*

Base class for all project items. Create root and category items by instantiating objects from this class.

name
Object name

Type str

description
Object description

Type str

is_root
True if new item should be a root item

Type bool

is_category
True if new item should be a category item

Type bool

parent (*self*)
Returns parent project item.

child_count (*self*)
Returns the number of child project items for this object.

children (*self*)
Returns the children of this project item.

child (*self*, *row*)
Returns child ProjectItem on given row.

Parameters **row** (*int*) – Row of child to return

Returns ProjectItem on given row or None if it does not exist

row(*self*)
Returns the row on which this project item is located.

add_child(*self*, *child_item*)
Append child project item as the last item in the children list. Set parent of this items parent as this item. This method is called by ProjectItemModel when new items are added.

Parameters **child_item** (ProjectItem) – Project item to add

Returns True if operation succeeded, False otherwise

remove_child(*self*, *row*)
Remove the child of this ProjectItem from given row. Do not call this method directly. This method is called by ProjectItemModel when items are removed.

Parameters **row** (*int*) – Row of child to remove

Returns True if operation succeeded, False otherwise

connect_signals(*self*)
Connect signals to handlers.

disconnect_signals(*self*)
Disconnect signals from handlers and check for errors.

16.7 ui_main

Contains ToolboxUI class.

author

P. Savolainen (VTT)

date 14.12.2017

16.7.1 Module Contents

class ui_main.**ToolboxUI**
Bases: PySide2.QtWidgets.QMainWindow

Class for application main GUI functions.

msg

msg_success

msg_error

msg_warning

msg_proc

msg_proc_error

connect_signals(*self*)
Connect signals.

project(*self*)
Returns current project or None if no project open.

qsettings(*self*)
Returns application preferences object.

init_project (*self*)
Initializes project at application start-up. Loads the last project that was open when app was closed or starts without a project if app is started for the first time.

new_project (*self*)
Shows new project form.

create_project (*self*, *name*, *description*)
Create new project and set it active.

Parameters

- **name** (*str*) – Project name
- **description** (*str*) – Project description

open_project (*self*, *load_path=None*)
Load project from a save file (.proj) file.

Parameters

- **load_path** (*str*) – Path to project save file. If default value is used,
- **file explorer dialog is opened where the user can select the** (*a*) –
- **file to load.** (*project*) –

Returns True when opening the project succeeded, False otherwise

Return type bool

save_project (*self*)
Save project.

save_project_as (*self*)
Ask user for a new project name and save. Creates a duplicate of the open project.

init_models (*self*, *tool_template_paths*)
Initialize application internal data models.

Parameters **tool_template_paths** (*list*) – List of tool definition file paths used in this project

init_project_item_model (*self*)
Initializes project item model. Create root and category items and add them to the model.

init_tool_template_model (*self*, *tool_template_paths*)
Initializes Tool template model.

Parameters **tool_template_paths** (*list*) – List of tool definition file paths used in this project

init_connection_model (*self*)
Initializes a model representing connections between project items.

init_shared_widgets (*self*)
Initialize widgets that are shared among all ProjectItems of the same type.

restore_ui (*self*)
Restore UI state from previous session.

clear_ui (*self*)
Clean UI to make room for a new or opened project.

item_selection_changed(*self*, *selected*, *deselected*)
Synchronize selection with scene. Check if only one item is selected and make it the active item: disconnect signals of previous active item, connect signals of current active item and show correct properties tab for the latter.

activate_no_selection_tab(*self*)
Shows 'No Selection' tab.

activate_item_tab(*self*, *item*)
Shows project item properties tab according to item type. Note: Does not work if a category item is given as argument.

Parameters **item** (ProjectItem) – Instance of a project item

open_tool_template(*self*)
Open a file dialog so the user can select an existing tool template .json file. Continue loading the tool template into the Project if successful.

add_tool_template(*self*, *tool_template*)
Add a ToolTemplate instance to project, which then can be added to a Tool item. Add tool template definition file path into project file (.proj)

tool_template (ToolTemplate): Tool template that is added to project

update_tool_template(*self*, *row*, *tool_template*)
Update a Tool template and refresh Tools that use it.

Parameters

- **row** (*int*) – Row of tool template in ToolTemplateModel
- **tool_template** (ToolTemplate) – An updated Tool template

remove_selected_tool_template(*self*)
Prepare to remove tool template selected in QListView.

remove_tool_template(*self*, *index*)
Remove tool template from ToolTemplateModel and tool definition file path from project file. Removes also Tool templates from all Tool items that use this template.

remove_all_items(*self*)
Slot for Remove All button.

remove_item(*self*, *ind*, *delete_item=False*, *check_dialog=False*)
Removes item from project when it's index in the project model is known. To remove all items in project, loop all indices through this method. This method is used in both opening and creating a new project as well as when item(s) are deleted from project. Use delete_item=False when closing the project or creating a new one. Setting delete_item=True deletes the item irrevocably. This means that data directories will be deleted from the hard drive. Handles also removing the node from the dag graph that contains it.

Parameters

- **ind** (*QModelIndex*) – Index of removed item in project model
- **delete_item** (*bool*) – If set to True, deletes the directories and data associated with the item
- **check_dialog** (*bool*) – If True, shows 'Are you sure?' message box

open_anchor(*self*, *qurl*)
Open file explorer in the directory given in qurl.

Parameters **qurl** (*QUrl*) – Directory path or a file to open

edit_tool_template(*self*, *index*)
Open the tool template widget for editing an existing tool template.

Parameters **index** (*QModelIndex*) – Index of the item (from double-click or contex menu signal)

open_tool_template_file(*self*, *index*)
Open the Tool template definition file in the default (.json) text-editor.

Parameters **index** (*QModelIndex*) – Index of the item

open_tool_main_program_file(*self*, *index*)
Open the tool template's main program file in the default editor.

Parameters **index** (*QModelIndex*) – Index of the item

export_as_graphml(*self*)
Exports all DAGs in project to separate GraphML files.

connection_data_changed(*self*, *index*)
[OBSOLETE?] Called when checkbox delegate wants to edit connection data. Add or remove Link instance accordingly.

_handle_zoom_widget_minus_pressed(*self*)
Slot for handling case when '-' button in menu is pressed.

_handle_zoom_widget_plus_pressed(*self*)
Slot for handling case when '+' button in menu is pressed.

_handle_zoom_widget_reset_pressed(*self*)
Slot for handling case when 'reset zoom' button in menu is pressed.

setup_zoom_action(*self*)
Setup zoom action in view menu.

restore_dock_widgets(*self*)
Dock all floating and or hidden QDockWidgets back to the main window.

set_debug_qactions(*self*)
Set shortcuts for QActions that may be needed in debugging.

hide_tabs(*self*)
Hides project item info tab bar and connections tab in project item QTreeView. Makes (hidden) actions on how to show them if needed for debugging purposes.

add_toggle_view_actions(*self*)
Add toggle view actions to View menu.

toggle_tabbar_visibility(*self*)
Shows or hides the tab bar in project item info tab widget. For debugging purposes.

toggle_connections_tab_visibility(*self*)
Shows or hides connections tab in the project item QTreeView. For debugging purposes.

update_datetime(*self*)
Returns a boolean, which determines whether date and time is prepended to every Event Log message.

add_message(*self*, *msg*)
Append regular message to Event Log.

Parameters **msg** (*str*) – String written to QTextBrowser

add_success_message(*self*, *msg*)
Append message with green text color to Event Log.

Parameters **msg** (*str*) – String written to QTextBrowser

add_error_message(*self*, *msg*)
Append message with red color to Event Log.

Parameters **msg** (*str*) – String written to QTextBrowser

add_warning_message(*self*, *msg*)
Append message with yellow (golden) color to Event Log.

Parameters **msg** (*str*) – String written to QTextBrowser

add_process_message(*self*, *msg*)
Writes message from stdout to process output QTextBrowser.

Parameters **msg** (*str*) – String written to QTextBrowser

add_process_error_message(*self*, *msg*)
Writes message from stderr to process output QTextBrowser.

Parameters **msg** (*str*) – String written to QTextBrowser

show_add_data_store_form(*self*, *x=0*, *y=0*)
Show add data store widget.

show_add_data_connection_form(*self*, *x=0*, *y=0*)
Show add data connection widget.

show_add_data_interface_form(*self*, *x=0*, *y=0*)
Show add data interface widget.

show_add_tool_form(*self*, *x=0*, *y=0*)
Show add tool widget.

show_add_view_form(*self*, *x=0*, *y=0*)
Show add view widget.

show_tool_template_form(*self*, *tool_template=None*)
Show create tool template widget.

show_settings(*self*)
Show Settings widget.

show_tool_config_asst(*self*)
Show Tool configuration assistant widget.

show_about(*self*)
Show About Spine Toolbox form.

show_user_guide(*self*)
Open Spine Toolbox documentation index page in browser.

show_getting_started_guide(*self*)
Open Spine Toolbox Getting Started HTML page in browser.

show_item_context_menu(*self*, *pos*)
Context menu for project items listed in the project QTreeView.

Parameters **pos** (*QPoint*) – Mouse position

show_item_image_context_menu(*self*, *pos*, *name*)
Context menu for project item images on the QGraphicsView.

Parameters

- **pos** (*QPoint*) – Mouse position

- **name** (*str*) – The name of the concerned item

show_project_item_context_menu(*self*, *pos*, *ind*)
Create and show project item context menu.

Parameters

- **pos** (*QPoint*) – Mouse position
- **ind** (*QModelIndex*) – Index of concerned item

show_link_context_menu(*self*, *pos*, *link*)
Context menu for connection links.

Parameters

- **pos** (*QPoint*) – Mouse position
- **link** (Link *(QGraphicsPathItem)*) – The concerned link

show_tool_template_context_menu(*self*, *pos*)
Context menu for tool templates.

Parameters **pos** (*QPoint*) – Mouse position

show_dc_ref_properties_context_menu(*self*, *pos*)
Create and show a context-menu in data connection properties references view.

Parameters **pos** (*QPoint*) – Mouse position

show_dc_data_properties_context_menu(*self*, *pos*)
Create and show a context-menu in data connection properties data view.

Parameters **pos** (*QPoint*) – Mouse position

show_tool_properties_context_menu(*self*, *pos*)
Create and show a context-menu in Tool properties if selected Tool has a Tool template.

Parameters **pos** (*QPoint*) – Mouse position

show_view_properties_context_menu(*self*, *pos*)
Create and show a context-menu in View properties.

Parameters **pos** (*QPoint*) – Mouse position

show_di_files_properties_context_menu(*self*, *pos*)
Create and show a context-menu in Data Interface properties source files view.

Parameters **pos** (*QPoint*) – Mouse position

remove_refs_with_del_key(*self*)
Slot that removes selected references from the currently selected Data Connection. Used when removing DC references by pressing the Delete key on keyboard (Qt.Key_Delete).

remove_data_with_del_key(*self*)
Slot that removes selected data files from the currently selected Data Connection. Used when removing DC data files by pressing the Delete key on keyboard (Qt.Key_Delete).

close_view_forms(*self*)
Closes all GraphViewForm, TreeViewForm, and TabularViewForm instances opened in Data Stores and Views. Ensures that close() method is called on all corresponding DiffDatabaseMapping instances, which cleans up the databases. Also closes all SpineDatapackageWidget instances opened in Data Connections.

show_confirm_exit(*self*)
Shows confirm exit message box.

Returns True if user clicks Yes or False if exit is cancelled

show_save_project_prompt(*self*)
Shows the save project message box.

closeEvent(*self*, *event=None*)
Method for handling application exit.

Parameters **event** (*QEvent*) – PySide2 event

16.8 data_connection

Module for data connection class.

author

P. Savolainen (VTT)

date 19.12.2017

16.8.1 Module Contents

class data_connection.**DataConnection**(*toolbox*, *name*, *description*, *references*, *x*, *y*)
Bases: *project_item.ProjectItem*

Data Connection class.

toolbox
QMainWindow instance

Type *ToolboxUI*

name
Object name

Type str

description
Object description

Type str

references
List of file references

Type list

x
Initial X coordinate of item icon

Type int

y
Initial Y coordinate of item icon

Type int

make_signal_handler_dict(*self*)
Returns a dictionary of all shared signals and their handlers. This is to enable simpler connecting and disconnecting.

activate(*self*)
Restore selections and connect signals.

deactivate(*self*)
Save selections and disconnect signals.

restore_selections(*self*)
Restore selections into shared widgets when this project item is selected.

save_selections(*self*)
Save selections in shared widgets for this project item into instance variables.

get_icon(*self*)
Returns the item representing this data connection in the scene.

add_files_to_references(*self*, *paths*)
Add multiple file paths to reference list.

Parameters **paths** (*list*) – A list of paths to files

receive_files_dropped_on_dc(*self*, *item*, *file_paths*)
Called when files are dropped onto a data connection graphics item. If the item is this Data Connection's graphics item, add the files to data.

add_files_to_data_dir(*self*, *file_paths*)
Add files to data directory

open_directory(*self*, *checked=False*)
Open file explorer in Data Connection data directory.

add_references(*self*, *checked=False*)
Let user select references to files for this data connection.

remove_references(*self*, *checked=False*)
Remove selected references from reference list. Do not remove anything if there are no references selected.

copy_to_project(*self*, *checked=False*)
Copy selected file references to this Data Connection's data directory.

open_reference(*self*, *index*)
Open reference in default program.

open_data_file(*self*, *index*)
Open data file in default program.

show_spine_datapackage_form(*self*)
Show spine_datapackage_form widget.

datapackage_form_destroyed(*self*)
Notify a connection that datapackage form has been destroyed.

make_new_file(*self*)
Create a new blank file to this Data Connections data directory.

remove_files(*self*)
Remove selected files from data directory.

file_references(*self*)
Returns a list of paths to files that are in this item as references.

data_files(*self*)
Returns a list of files that are in the data directory.

refresh (*self*)
Refresh data files in Data Connection Properties. NOTE: Might lead to performance issues.

populate_reference_list (*self*, *items*)
List file references in QTreeView. If items is None or empty list, model is cleared.

populate_data_list (*self*, *items*)
List project internal data (files) in QTreeView. If items is None or empty list, model is cleared.

update_name_label (*self*)
Update Data Connection tab name label. Used only when renaming project items.

execute (*self*)
Executes this Data Connection.

stop_execution (*self*)
Stops executing this Data Connection.

16.9 plotting

Functions for plotting on PlotWidget.

Currently plotting from the table views found in Graph, Tree and Tabular views are supported.

The main entrance points to plotting are: - plot_selection() which plots selected cells on a table view returning a PlotWidget object - plot_pivot_column() which is a specialized method for plotting entire columns of a pivot table - add_time_series_plot() which adds a time series plot to an existing PlotWidget

author

A. Soininen(VTT)

date 9.7.2019

16.9.1 Module Contents

exception plotting.**PlottingError**(*message*)
Bases: Exception

An exception signalling failure in plotting.

message
an error message

Type str

message
Returns the error message.

plotting.**_add_plot_to_widget**(*values*, *labels*, *plot_widget*)
Adds a new plot to plot_widget.

plotting.**_raise_if_types_inconsistent**(*values*)
Raises an exception if not all values are TimeSeries or floats.

plotting.**_filter_name_columns**(*selections*)
Returns a dict with all but the entry with the greatest key removed.

plotting.**_organize_selection_to_columns**(*indexes*)
Organizes a list of model indexes into a dictionary of {column: (rows)} entries.

plotting.**_collect_single_column_values**(*model*, *column*, *rows*, *hints*)
Collects selected parameter values from a single column in a PivotTableModel.

The return value of this function depends on what type of data the given column contains. In case of plain numbers, a list of floats and a single label string are returned. In case of time series, a list of TimeSeries objects is returned, accompanied by a list of labels, each label corresponding to one of the time series.

Parameters

- **model** (*QAbstractTableModel*) – a table model
- **column** (*int*) – a column index to the model
- **rows** (*Sequence*) – row indexes to plot
- **hints** (PlottingHints) – a plot support object

Returns a tuple of values and label(s)

plotting.**_collect_column_values**(*model*, *column*, *rows*, *hints*)
Collects selected parameter values from a single column in a PivotTableModel for plotting.

The return value of this function depends on what type of data the given column contains. In case of plain numbers, a single tuple of two lists of x and y values and a single label string are returned. In case of time series, a list of TimeSeries objects is returned, accompanied by a list of labels, each label corresponding to one of the time series.

Parameters

- **model** (*QAbstractTableModel*) – a table model
- **column** (*int*) – a column index to the model
- **rows** (*Sequence*) – row indexes to plot
- **hints** (PlottingHints) – a support object

Returns a tuple of values and label(s)

plotting.**plot_pivot_column**(*model*, *column*, *hints*)
Returns a plot widget with a plot of an entire column in PivotTableModel.

Parameters

- **model** (PivotTableModel) – a pivot table model
- **column** (*int*) – a column index to the model
- **hints** (PlottingHints) – a helper needed for e.g. plot labels

Returns a PlotWidget object

plotting.**plot_selection**(*model*, *indexes*, *hints*)
Returns a plot widget with plots of the selected indexes.

Parameters

- **model** (*QAbstractTableModel*) – a model
- **indexes** (*Iterable*) – a list of QModelIndex objects for plotting
- **hints** (PlottingHints) – a helper needed for e.g. plot labels

Returns a PlotWidget object

plotting.**add_time_series_plot**(*plot_widget*, *value*, *label=None*)
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
- **label** (*str*) – a label for the time series

plotting.**tree_graph_view_parameter_value_name**(*index*, *table_view*)
Returns a label for Tree or Graph view table cell.

Parameters

- **index** (*QModelIndex*) – an index to the table model
- **table_view** (*QTableView*) – a table view widget corresponding to index

Returns a unique name for the parameter value as a string

class plotting.**PlottingHints**
A base class for plotting hints.

The functionality in this class allows the plotting functions to work without explicit knowledge of the underlying table model or widget.

cell_label(*self*, *model*, *index*)
Returns a label for the cell given by index in a table.

column_label(*self*, *model*, *column*)
Returns a label for a column.

filter_columns(*self*, *selections*, *model*)
Filters columns and returns the filtered selections.

is_index_in_data(*self*, *model*, *index*)
Returns true if the cell given by index is actually plottable data.

special_x_values(*self*, *model*, *column*, *rows*)
Returns X values if available, otherwise returns None.

x_label(*self*, *model*)
Returns a label for the x axis.

class plotting.**GraphAndTreeViewPlottingHints**(*table_view*)
Bases: *plotting.PlottingHints*

Support for plotting data in Graph and Tree views.

table_view
a parameter value or definition widget

Type QTableView

cell_label(*self*, *model*, *index*)
Returns a label build from the columns on the left from the data column.

column_label(*self*, *model*, *column*)
Returns the column header.

filter_columns(*self*, *selections*, *model*)
Returns the 'value' or 'default_value' column only.

is_index_in_data(*self*, *model*, *index*)
Always returns True.

special_x_values (*self*, *model*, *column*, *rows*)
Always returns None.

x_label (*self*, *model*)
Returns an empty string for the x axis label.

class plotting.**PivotTablePlottingHints**
Bases: *plotting.PlottingHints*

Support for plotting data in Tabular view.

cell_label (*self*, *model*, *index*)
Returns a label for the table cell given by index.

column_label (*self*, *model*, *column*)
Returns a label for a table column.

filter_columns (*self*, *selections*, *model*)
Filters the X column from selections.

is_index_in_data (*self*, *model*, *index*)
Returns True if index is in the data portion of the table.

special_x_values (*self*, *model*, *column*, *rows*)
Returns the values from the X column if one is designated otherwise returns None.

x_label (*self*, *model*)
Returns the label of the X column, if available.

16.10 tool_configuration_assistants

Classes for tool configuration assistants.

authors

M. Marin (KTH)

date 10.1.2019

16.10.1 Module Contents

class tool_configuration_assistants.**SpineModelConfigurationAssistant** (*toolbox*)
Bases: PySide2.QtCore.QObject

Configuration assistant for SpineModel.jl.

toolbox
QMainWindow instance

Type *ToolboxUI*

check_finished

installation_finished

msg

find_out_julia_version_and_project (*self*)

julia_version (*self*)
Return current julia version.

julia_active_project (*self*)
Return current julia active project.

spine_model_version_check (*self*)
Return qsubprocess that checks current version of SpineModel.

py_call_program_check (*self*)
Return qsubprocess that checks the python program used by PyCall in current julia version.

install_spine_model (*self*)
Return qsubprocess that installs SpineModel in current julia version.

install_py_call (*self*)
Return qsubprocess that installs PyCall in current julia version.

reconfigure_py_call (*self*, *pyprogramname*)
Return qsubprocess that reconfigure PyCall to use given python program.

16.11 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

16.11.1 Module Contents

class time_series_model_variable_resolution.**TimeSeriesModelVariableResolution**(*series*)
Bases: *indexed_value_table_model.IndexedValueTableModel*

A model for variable resolution time series type parameter values.

series
a time series

Type TimeSeriesVariableResolution

indexes
Returns the time stamps as an array.

values
Returns the values of the time series as an array.

data (*self*, *index*, *role=Qt.DisplayRole*)
Returns the time stamp or the corresponding value at given model index.

Column index 0 refers to time stamps while index 1 to values.

Parameters

- **index** (*QModelIndex*) – an index to the model
- **role** (*int*) – a role

flags (*self*, *index*)
Returns the flags for given model index.

insertRows (*self*, *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

removeRows (*self*, *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.

reset (*self*, *value*)
Resets the model with new time series data.

setData (*self*, *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

batch_set_data (*self*, *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 (*self*, *ignore_year*)
Sets the ignore_year option of the time series.

set_repeat (*self*, *repeat*)
Sets the repeat option of the time series.

16.12 executioner

Contains classes for handling project item execution.

author

P. Savolainen (VTT)

date 8.4.2019

16.12.1 Module Contents

class executioner.**DirectedGraphHandler**(*toolbox*)

Class for manipulating graphs according to user's actions.

Parameters **toolbox** (ToolboxUI) – QMainWindow instance

dags(*self*)

Returns a list of graphs (DiGraph) in the project.

add_dag(*self*, *dag*)

Add graph to list.

Parameters **dag** (*DiGraph*) – Graph to add

remove_dag(*self*, *dag*)

Remove graph from instance variable list.

Parameters **dag** (*DiGraph*) – Graph to remove

add_dag_node(*self*, *node_name*)

Create directed graph with one node and add it to list.

Parameters **node_name** (*str*) – Project item name to add as a node

add_graph_edge(*self*, *src_node*, *dst_node*)

Adds an edge between the src and dst nodes. If nodes are in different graphs, the reference to union graph is saved and the references to the original graphs are removed. If src and dst nodes are already in the same graph, the edge is added to the graph. If src and dst are the same node, a self-loop (feedback) edge is added.

Parameters

- **src_node** (*str*) – Source project item node name
- **dst_node** (*str*) – Destination project item node name

remove_graph_edge(*self*, *src_node*, *dst_node*)

Removes edge from a directed graph.

Parameters

- **src_node** (*str*) – Source project item node name
- **dst_node** (*str*) – Destination project item node name

remove_node_from_graph(*self*, *node_name*)

Removes node from a graph that contains it. Called when project item is removed from project.

Parameters **node_name** (*str*) – Project item name

rename_node(*self*, *old_name*, *new_name*)

Handles renaming the node and edges in a graph when a project item is renamed.

Parameters

- **old_name** (*str*) – Old project item name
- **new_name** (*str*) – New project item name

Returns True if successful, False if renaming failed

Return type bool

dag_with_node(*self*, *node_name*)
Returns directed graph that contains given node.

Parameters **node_name** (*str*) – Node to look for

Returns Directed graph that contains node or None if not found.

Return type (DiGraph)

dag_with_edge(*self*, *src_node*, *dst_node*)
Returns directed graph that contains given edge.

Parameters

- **src_node** (*str*) – Source node name
- **dst_node** (*str*) – Destination node name

Returns Directed graph that contains edge or None if not found.

Return type (DiGraph)

calc_exec_order(*self*, *g*)
Returns an bfs-ordered list of nodes in the given graph. Adds a dummy source node to the graph if there are more than one nodes that have no inbound connections. The dummy source node is needed for the bfs-algorithm.

Parameters **g** (*DiGraph*) – Directed graph to process

Returns bfs-ordered list of node names (first item at index 0). Empty list if given graph is not a DAG.

Return type list

node_is_isolated(*self*, *node*, *allow_self_loop=False*)
Checks if the project item with the given name has any connections.

Parameters

- **node** (*str*) – Project item name
- **allow_self_loop** (*bool*) – If default (False), Self-loops are considered as an in-neighbor or an out-neighbor so the method returns False. If True, single node with a self-loop is considered isolated.

Returns

True if project item has no in-neighbors nor out-neighbors, False if it does. Single node with a self-loop is NOT isolated (returns False).

Return type bool

static source_nodes(*g*)
Returns a list of source nodes in given graph. A source node has no incoming edges. This is determined by calculating the in-degree of each node in the graph. If nodes in-degree == 0, it is a source node

Parameters **g** (*DiGraph*) – Graph to examine

Returns List of source node names or an empty list is there are none.

Return type list

static nodes_connected(*dag*, *a*, *b*)
Checks if node a is connected to node b. Edge directions are ignored. If any of source node a's ancestors or descendants have a path to destination node b, returns True. Also returns True if destination node b has a path to any of source node a's ancestors or descendants.

Parameters

- **dag** (*DiGraph*) – Graph that contains nodes a and b
- **a** (*str*) – Node name
- **b** (*str*) – Another node name

Returns True if a and b are connected, False otherwise

Return type bool

static export_to_graphml(*g*, *path*)
Export given graph to a path in GraphML format.

Parameters

- **g** (*DiGraph*) – Graph to export
- **path** (*str*) – Full output path for GraphML file

Returns Operation success status

Return type bool

class executioner.**ExecutionInstance**(*toolbox*, *execution_list*)
Bases: PySide2.QtCore.QObject

Class for the graph that is being executed. Contains references to files and resources advertised by project items so that project items downstream can find them.

Parameters

- **toolbox** (ToolboxUI) – QMainWindow instance
- **execution_list** (*list*) – Ordered list of nodes to execute

graph_execution_finished_signal

project_item_execution_finished_signal

start_execution(*self*)
Pops the next item from the execution list and starts executing it.

execute_project_item(*self*)
Starts executing project item.

item_execution_finished(*self*, *item_finish_state*)
Pop next project item to execute or finish current graph if there are no items left.

Parameters

- **item_finish_state** (*int*) – 0=Continue to next project item. -2=Stop executing this graph (happens when e.g.
- **does not find req. input files or something)** (Tool) –

stop(*self*)
Stops running project item and terminates current graph execution.

add_ds_ref (*self*, *dialect*, *ref*)
Adds given database reference to a dictionary. Key is the dialect. If dialect is sqlite, value is a list of full paths to sqlite files. For other dialects, key is the dialect and value is a list of URLs to database servers.

Parameters

- **dialect** (*str*) – Dialect name (lower case)
- **ref** (*str*) – Database reference

add_di_data (*self*, *di_name*, *data*)
Adds given data from data interface to a list.

Parameters

- **di_name** (*str*) – Data interface name
- **data** (*dict*) – Data to import

append_dc_refs (*self*, *refs*)
Adds given file paths (Data Connection file references) to a list.

Parameters **refs** (*list*) – List of file paths (references)

append_dc_files (*self*, *files*)
Adds given project data file paths to a list.

Parameters **files** (*list*) – List of file paths

append_tool_output_file (*self*, *filepath*)
Adds given file path to a list containing paths to Tool output files.

Parameters **filepath** (*str*) – Path to a tool output file (in tool result directory)

find_file (*self*, *filename*)
Returns the first occurrence to full path to given file name or None if file was not found.

Parameters **filename** (*str*) – Searched file name (no path) TODO: Change to pattern

Returns Full path to file if found, None if not found

Return type str

find_optional_files (*self*, *pattern*)
Returns a list of found paths to files that match the given pattern.

Returns List of (full) paths

Return type list

16.13 indexed_value_table_model

A model for indexed parameter values, used by the parameter value editors.

authors

A. Soininen (VTT)

date 18.6.2019

16.13.1 Module Contents

class indexed_value_table_model.**IndexedValueTableModel**(*value*, *index_header*, *value_header*)

Bases: PySide2.QtCore.QAbstractTableModel

A base class for time pattern and time series models.

value
a parameter value

Type TimePattern, TimeSeriesFixedStep, TimeSeriesVariableStep

index_header
a header for the index column

Type str

value_header
a header for the value column

Type str

value
Returns the parameter value associated with the model.

columnCount(*self*, *parent=QModelIndex()*)
Returns the number of columns which is two.

data(*self*, *index*, *role=Qt.DisplayRole*)
Returns the data at index for given role.

headerData(*self*, *section*, *orientation=Qt.Horizontal*, *role=Qt.DisplayRole*)
Returns a header.

reset(*self*, *value*)
Resets the model.

rowCount(*self*, *parent=QModelIndex()*)
Returns the number of rows.

16.14 spinetoolbox

Spine Toolbox application main file.

author

P. Savolainen (VTT)

date 14.12.2017

16.14.1 Module Contents

spinetoolbox.**main**(*argv*)
Launch application.

Parameters **argv** (*list*) – Command line arguments

16.15 helpers

General helper functions and classes.

authors

P. Savolainen (VTT)

date 10.1.2018

16.15.1 Module Contents

helpers.**set_taskbar_icon**()
Set application icon to Windows taskbar.

helpers.**supported_img_formats**()
Function to check if reading .ico files is supported.

helpers.**pyside2_version_check**()
Check that PySide2 version is older than 5.12, since this is not supported yet. Issue #238 in GitLab.

qt_version is the Qt version used to compile PySide2 as string. E.g. "5.11.2" qt_version_info is a tuple with each version component of Qt used to compile PySide2. E.g. (5, 11, 2)

helpers.**spinedb_api_version_check**()
Check if spinedb_api is the correct version and explain how to upgrade if it is not.

helpers.**busy_effect**(*func*)
Decorator to change the mouse cursor to 'busy' while a function is processed.

Parameters **func** – Decorated function.

helpers.**project_dir**(*qsettings*)
Returns current project directory.

Parameters **qsettings** (*QSettings*) – Settings object

helpers.**get_datetime**(*show*)
Returns date and time string for appending into Event Log messages.

Parameters **show** (*boolean*) – True returns date and time string. False returns empty string.

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.

Returns True if directory already exists or if it was created successfully.

Raises OSError if operation failed.

helpers.**create_output_dir_timestamp**()
Creates a new timestamp string that is used as Tool output directory.

Returns Timestamp string or empty string if failed.

helpers.**create_log_file_timestamp**()
Creates a new timestamp string that is used as Data Interface and Data Store error log file.

Returns Timestamp string or empty string if failed.

helpers.**copy_files**(*src_dir*, *dst_dir*, *includes=None*, *excludes=None*)
Method for copying files. Does not copy folders.

Parameters

- **src_dir** (*str*) – Source directory
- **dst_dir** (*str*) – Destination directory
- **includes** (*list*) – Included files (wildcards accepted)
- **excludes** (*list*) – Excluded files (wildcards accepted)

Returns Number of files copied

Return type count (int)

helpers.**erase_dir**(*path*, *verbosity=False*)
Delete directory and all its contents without prompt.

Parameters

- **path** (*str*) – Path to directory
- **verbosity** (*bool*) – Print logging messages or not

helpers.**copy_dir**(*widget*, *src_dir*, *dst_dir*)
Make a copy of a directory. All files and folders are copied.

Parameters

- **widget** (*QWidget*) – Parent widget for QMessageBoxes
- **src_dir** (*str*) – Absolute path to directory that will be copied
- **dst_dir** (*str*) – Absolute path to new directory

helpers.**rename_dir**(*widget*, *old_dir*, *new_dir*)
Rename directory. Note: This is not used in renaming projects due to unreliability. Looks like it works fine in renaming project items though.

Parameters

- **widget** (*QWidget*) – Parent widget for QMessageBoxes
- **old_dir** (*str*) – Absolute path to directory that will be renamed
- **new_dir** (*str*) – Absolute path to new directory

helpers.**fix_name_ambiguity**(*name_list*, *offset=0*)
Modify repeated entries in name list by appending an increasing integer.

helpers.**tuple_itemgetter**(*itemgetter_func*, *num_indexes*)
Change output of itemgetter to always be a tuple even for one index

helpers.**format_string_list**(*str_list*)
Return an unordered html list with all elements in str_list. Intended to print error logs as returned by spinedb_api.

Parameters **str_list** (*list(str)*) –

helpers.**get_db_map**(*url*, *upgrade=False*)
Returns a DiffDatabaseMapping instance from url. If the db is not the latest version, asks the user if they want to upgrade it.

helpers.**do_get_db_map**(*url*, *upgrade*)
Returns a DiffDatabaseMapping instance from url. Called by *get_db_map*.

helpers.**int_list_to_row_count_tuples**(*int_list*)
Breaks a list of integers into a list of tuples (row, count) corresponding to chunks of successive elements.

class helpers.**IconListManager**(*icon_size*)
A class to manage icons for icon list widgets.

init_model(*self*)
Init model that can be used to display all icons in a list.

_model_data(*self*, *index*, *role*)
Replacement method for model.data().

Create pixmaps as they're requested by the data() method, to reduce loading time.

create_object_pixmap(*self*, *display_icon*)
Create and return a pixmap corresponding to display_icon.

class helpers.**IconManager**
A class to manage object class icons for data store forms.

ICON_SIZE

create_object_pixmap(*self*, *display_icon*)
Create a pixmap corresponding to display_icon, cache it, and return it.

setup_object_pixmaps(*self*, *object_classes*)
Called after adding or updating object classes. Create the corresponding object pixmaps and clear obsolete entries from the relationship class icon cache.

object_pixmap(*self*, *object_class_name*)
A pixmap for the given object class.

object_icon(*self*, *object_class_name*)
An icon for the given object class.

relationship_pixmap(*self*, *str_object_class_name_list*)
A pixmap for the given object class name list, created by rendering several object pixmaps next to each other.

relationship_icon(*self*, *str_object_class_name_list*)
An icon for the given object class name list.

class helpers.**CharIconEngine**(*char*, *color*)
Bases: PySide2.QtGui.QIconEngine

Specialization of QIconEngine used to draw font-based icons.

paint(*self*, *painter*, *rect*, *mode=None*, *state=None*)

pixmap(*self*, *size*, *mode=None*, *state=None*)

helpers.**make_icon_id**(*icon_code*, *color_code*)
Take icon and color codes, and return equivalent integer.

helpers.**interpret_icon_id**(*display_icon*)
Take a display icon integer and return an equivalent tuple of icon and color code.

helpers.**default_icon_id**()

16.16 tool

Tool class.

author

P. Savolainen (VTT)

date 19.12.2017

16.16.1 Module Contents

class tool.**Tool**(*toolbox*, *name*, *description*, *tool_template*, *use_work*, *x*, *y*)

Bases: *project_item.ProjectItem*

Tool class.

toolbox
QMainWindow instance

Type *ToolboxUI*

name
Object name

Type str

description
Object description

Type str

tool_template
Template for this Tool

Type *ToolTemplate*

use_work
Execute associated Tool template in work (True) or source directory (False)

Type bool

x
Initial X coordinate of item icon

Type int

y
Initial Y coordinate of item icon

Type int

make_signal_handler_dict(*self*)
Returns a dictionary of all shared signals and their handlers. This is to enable simpler connecting and disconnecting.

activate(*self*)
Restore selections and connect signals.

deactivate(*self*)
Save selections and disconnect signals.

restore_selections (*self*)
Restore selections into shared widgets when this project item is selected.

save_selections (*self*)
Save selections in shared widgets for this project item into instance variables.

update_execution_mode (*self*, *checked*)
Slot for execute in work radio button toggled signal.

update_tool_template (*self*, *row*)
Update Tool template according to selection in the template comboBox.

Parameters **row** (*int*) – Selected row in the comboBox

set_tool_template (*self*, *tool_template*)
Sets Tool Template for this Tool. Removes Tool Template if None given as argument.

Parameters **tool_template** (ToolTemplate) – Template for this Tool. None removes the template.

update_tool_ui (*self*)
Update Tool UI to show Tool template details. Used when Tool template is changed. Overrides execution mode (work or source) with the template default.

restore_tool_template (*self*, *tool_template*)
Restores the Tool Template of this Tool. Removes Tool Template if None given as argument. Needed in order to override tool template default execution mode (work or source).

Parameters **tool_template** (ToolTemplate) – Template for this Tool. None removes the template.

open_results (*self*, *checked=False*)
Open output directory in file browser.

get_icon (*self*)
Returns the graphics item representing this tool in the scene.

edit_tool_template (*self*)
Open Tool template editor for the Tool template attached to this Tool.

open_tool_template_file (*self*)
Open Tool template definition file.

open_tool_main_program_file (*self*)
Open Tool template main program file in an external text edit application.

open_tool_main_directory (*self*)
Open directory where the Tool template main program is located in file explorer.

tool_template (*self*)
Returns Tool template.

count_files_and_dirs (*self*)
Count the number of files and directories in required input files model.

Returns Tuple containing the number of required files and directories.

create_subdirectories (*self*)
Iterate items in required input files and check if there are any directories to create. Create found directories directly to ToolInstance base directory.

Returns Boolean variable depending on success

copy_input_files (*self*, *paths*)
Copy input files from given paths to work or source directory, depending on where the Tool template requires them to be.

Parameters **paths** (*dict*) – Key is path to destination file, value is path to source file.

Returns Boolean variable depending on operation success

copy_optional_input_files (*self*, *paths*)
Copy optional input files from given paths to work or source directory, depending on where the Tool template requires them to be.

Parameters **paths** (*dict*) – Key is the optional file name pattern, value is a list of paths to source files.

Returns Boolean variable depending on operation success

find_output_items (*self*)
Find output items of this Tool.

Returns List of Data Store and Data Connection items.

update_instance (*self*)
Initialize and update instance so that it is ready for processing. This is where Tool type specific initialization happens (whether the tool is GAMS, Python or Julia script).

append_instance_args (*self*)
Append Tool template command line args into instance args list.

get_instance_args (*self*)
Return instance args as list.

populate_source_file_model (*self*, *items*)
Add required source files (includes) into a model. If items is None or an empty list, model is cleared.

populate_input_file_model (*self*, *items*)
Add required Tool input files into a model. If items is None or an empty list, model is cleared.

populate_opt_input_file_model (*self*, *items*)
Add optional Tool template files into a model. If items is None or an empty list, model is cleared.

populate_output_file_model (*self*, *items*)
Add Tool output files into a model. If items is None or an empty list, model is cleared.

populate_template_model (*self*, *populate*)
Add all tool template specs to a single QTreeView. If items is None or an empty list, model is cleared.

Parameters **populate** (*bool*) – False to clear model, True to populate.

update_name_label (*self*)
Update Tool tab name label. Used only when renaming project items.

open_directory (*self*, *checked=False*)
Open file explorer in Tool data directory.

execute (*self*)
Executes this Tool.

find_input_files (*self*)
Iterates files in required input files model and looks for them from execution instance.

Returns Dictionary of paths where required files are found or None if some file was not found.

find_optional_input_files (*self*)
Tries to find optional input files from previous project items in the DAG. Returns found paths.

Returns Dictionary of optional input file paths or an empty dictionary if no files found. Key is the optional input item and value is a list of paths that matches the item.

execute_finished(*self*, *return_code*)
Tool template execution finished.

Parameters **return_code** (*int*) – Process exit code

stop_execution(*self*)
Stops executing this Tool.

stop_process(*self*, *checked=False*)
Terminate Tool template execution.

16.17 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

16.17.1 Module Contents

class time_series_model_fixed_resolution.**TimeSeriesModelFixedResolution**(*series*)
Bases: *indexed_value_table_model.IndexedValueTableModel*

A model for fixed resolution time series type parameter values.

series
a time series

Type TimeSeriesFixedResolution

indexes
Returns the time stamps as an array.

values
Returns the values of the time series as an array.

data(*self*, *index*, *role=Qt.DisplayRole*)
Returns the time stamp or the corresponding value at given model index.

Column index 0 refers to time stamps while index 1 to values.

Parameters

- **index** (*QModelIndex*) – an index to the model
- **role** (*int*) – a role

flags(*self*, *index*)
Returns flags at index.

insertRows(*self*, *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(*self*, *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(*self*, *value*)
Resets the model with new time series data.

setData(*self*, *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(*self*, *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(*self*, *ignore_year*)
Sets the ignore_year option of the time series.

set_repeat(*self*, *repeat*)
Sets the repeat option of the time series.

set_resolution(*self*, *resolution*)
Sets the resolution.

set_start(*self*, *start*)
Sets the start datetime.

16.18 datapackage_import_export

Functions to import/export between spine database and frictionless data's datapackage.

author

M. Marin (KTH)

date 28.8.2018

16.18.1 Module Contents

class datapackage_import_export.**Signaler**
Bases: PySide2.QtCore.QObject

finished

failed

progressed

class datapackage_import_export.**DatapackageToSpineConverter**(*db_url*, *datapackage_descriptor*, *datapackage_base_path*)
Bases: PySide2.QtCore.QRunnable

number_of_steps(*self*)

run(*self*)

_run(*self*)

datapackage_import_export.**datapackage_to_spine**(*db_map*, *datapackage_file_path*)
Convert datapackage from *datapackage_file_path* into Spine *db_map*.

16.19 treeview_models

Classes for handling models in tree and graph views.

authors

M. Marin (KTH)

date 28.6.2019

16.19.1 Module Contents

class treeview_models.**ObjectClassListModel**(*graph_view_form*)
Bases: PySide2.QtGui.QStandardItemModel

A class to list object classes in the GraphViewForm.

populate_list(*self*)
Populate model.

add_object_class(*self*, *object_class*)
Add object class item to model.

data(*self*, *index*, *role=Qt.DisplayRole*)
Returns the data stored under the given role for the item referred to by the index.

class treeview_models.**RelationshipClassListModel**(*graph_view_form*)
Bases: PySide2.QtGui.QStandardItemModel

A class to list relationship classes in the GraphViewForm.

populate_list(*self*)
Populate model.

add_relationship_class(*self*, *relationship_class*)
Add relationship class.

data(*self*, *index*, *role=Qt.DisplayRole*)
Returns the data stored under the given role for the item referred to by the index.

class treeview_models.**ObjectTreeModel**(*parent*, *flat=False*)
Bases: PySide2.QtGui.QStandardItemModel

A class to display Spine data structure in a treeview with object classes at the outer level.

data(*self*, *index*, *role=Qt.DisplayRole*)
Returns the data stored under the given role for the item referred to by the index.

static backward_sweep(*index*, *call=None*)
Sweep the tree from the given index towards the root, and apply *call* on each.

forward_sweep(*self*, *index*, *call=None*)
Sweep the tree from the given index towards the leaves, and apply *call* on each.

hasChildren(*self*, *parent*)
Return True if not fetched, so the user can try and expand it.

canFetchMore(*self*, *parent*)
Return True if not fetched.

fetchMore(*self*, *parent*)
Build the deeper levels of the tree

build_tree(*self*, *flat=False*)
Build the first level of the tree

new_object_class_row(*self*, *db_map*, *object_class*)
Returns new object class item.

new_object_row(*self*, *db_map*, *object_*)
Returns new object item.

new_relationship_class_row(*self*, *db_map*, *relationship_class*)
Returns new relationship class item.

new_relationship_row(*self*, *db_map*, *relationship*)
Returns new relationship item.

add_object_classes(*self*, *db_map*, *object_classes*)
Add object class items to given db.

add_objects(*self*, *db_map*, *objects*)
Add object items to the given db.

add_relationship_classes(*self*, *db_map*, *relationship_classes*)
Add relationship class items to model.

add_relationships (*self*, *db_map*, *relationships*)
Add relationship items to model.

add_objects_to_class (*self*, *db_map*, *objects*, *object_class_item*)

add_relationships_classes_to_object (*self*, *db_map*, *relationship_classes*, *object_item*)

add_relationships_to_class (*self*, *db_map*, *relationships*, *rel_cls_item*)

update_object_classes (*self*, *db_map*, *object_classes*)
Update object classes in the model. This of course means updating the object class name in relationship class items.

update_objects (*self*, *db_map*, *objects*)
Update object in the model. This of course means updating the object name in relationship items.

update_relationship_classes (*self*, *db_map*, *relationship_classes*)
Update relationship classes in the model.

update_relationships (*self*, *db_map*, *relationships*)
Update relationships in the model. Move rows if the objects in the relationship change.

remove_object_class_rows (*self*, *db_map*, *removed_rows*)

remove_object_rows (*self*, *db_map*, *removed_rows*, *object_class_item*)

remove_relationship_class_rows (*self*, *db_map*, *removed_rows*, *object_item*)

remove_relationship_rows (*self*, *db_map*, *removed_rows*, *rel_cls_item*)

remove_object_classes (*self*, *db_map*, *removed_ids*)
Remove object classes and their childs.

remove_objects (*self*, *db_map*, *removed_ids*)
Remove objects and their childs.

remove_relationship_classes (*self*, *db_map*, *removed_ids*)
Remove relationship classes and their childs.

remove_relationships (*self*, *db_map*, *removed_ids*)
Remove relationships.

next_relationship_index (*self*, *index*)
Find and return next ocurrence of relationship item.

class treeview_models.**RelationshipTreeModel** (*parent*)
Bases: PySide2.QtGui.QStandardItemModel

A class to display Spine data structure in a treeview with relationship classes at the outer level.

data (*self*, *index*, *role=Qt.DisplayRole*)
Returns the data stored under the given role for the item referred to by the index.

hasChildren (*self*, *parent*)
Return True if not fetched, so the user can try and expand it.

canFetchMore (*self*, *parent*)
Return True if not fetched.

fetchMore (*self*, *parent*)
Build the deeper level of the tree

build_tree (*self*)
Build the first level of the tree

new_relationship_class_row(*self*, *db_map*, *relationship_class*)
Returns new relationship class item.

new_relationship_row(*self*, *db_map*, *relationship*)
Returns new relationship item.

add_relationship_classes(*self*, *db_map*, *relationship_classes*)
Add relationship class items to the model.

add_relationships(*self*, *db_map*, *relationships*)
Add relationship items to model.

add_relationships_to_class(*self*, *db_map*, *relationships*, *rel_cls_item*)

update_object_classes(*self*, *db_map*, *object_classes*)
Update object classes in the model. This just means updating the object class name in relationship class items.

update_objects(*self*, *db_map*, *objects*)
Update object in the model. This just means updating the object name in relationship items.

update_relationship_classes(*self*, *db_map*, *relationship_classes*)
Update relationship classes in the model.

update_relationships(*self*, *db_map*, *relationships*)
Update relationships in the model.

remove_relationship_class_rows(*self*, *db_map*, *removed_rows*)

remove_relationship_rows(*self*, *db_map*, *removed_rows*, *rel_cls_item*)

remove_object_classes(*self*, *db_map*, *removed_ids*)
Remove object classes and their childs.

remove_objects(*self*, *db_map*, *removed_ids*)
Remove objects and their childs.

remove_relationship_classes(*self*, *db_map*, *removed_ids*)
Remove relationship classes and their childs.

remove_relationships(*self*, *db_map*, *removed_ids*)
Remove relationships.

class treeview_models.**SubParameterModel**(*parent*)
Bases: *models.MinimalTableModel*

A parameter model which corresponds to a slice of the entire table. The idea is to combine several of these into one big model. Allows specifying set of columns that are non-editable (e.g., object_class_name) TODO: how column insertion/removal impacts fixed_columns?

flags(*self*, *index*)
Make fixed indexes non-editable.

data(*self*, *index*, *role=Qt.DisplayRole*)
Paint background of fixed indexes gray.

batch_set_data(*self*, *indexes*, *data*)
Batch set data for indexes. Try and update data in the database first, and if successful set data in the model.

items_to_update(*self*, *indexes*, *data*)
A list of items (dict) to update in the database.

update_items_in_db(*self*, *items_to_update*)
A list of ids of items updated in the database.

class treeview_models.**SubParameterValueModel**(*parent*)
Bases: *treeview_models.SubParameterModel*

A parameter model which corresponds to a slice of an entire parameter value table. The idea is to combine several of these into one big model.

items_to_update(*self*, *indexes*, *data*)
A list of items (dict) for updating in the database.

update_items_in_db(*self*, *items_to_update*)
Try and update parameter values in database.

data(*self*, *index*, *role=Qt.DisplayRole*)
Limit the display of JSON data.

class treeview_models.**SubParameterDefinitionModel**(*parent*)
Bases: *treeview_models.SubParameterModel*

A parameter model which corresponds to a slice of an entire parameter definition table. The idea is to combine several of these into one big model.

items_to_update(*self*, *indexes*, *data*)
A list of items (dict) for updating in the database.

update_items_in_db(*self*, *items_to_update*)
Try and update parameter definitions in database.

data(*self*, *index*, *role=Qt.DisplayRole*)
Limit the display of JSON data.

class treeview_models.**EmptyParameterModel**(*parent*)
Bases: *models.EmptyRowModel*

An empty parameter model. It implements *bath_set_data* for all 'EmptyParameter' models.

batch_set_data(*self*, *indexes*, *data*)
Batch set data for indexes. Set data in model first, then check if the database needs to be updated as well. Extend set of indexes as additional data is set (for emitting dataChanged at the end).

items_to_add(*self*, *indexes*)

add_items_to_db(*self*, *items_to_add*)

class treeview_models.**EmptyParameterValueModel**(*parent*)
Bases: *treeview_models.EmptyParameterModel*

An empty parameter value model. Implements *add_items_to_db* for both EmptyObjectParameterValueModel and EmptyRelationshipParameterValueModel.

add_items_to_db(*self*, *items_to_add*)
Add parameter values to database.

class treeview_models.**EmptyObjectParameterValueModel**(*parent*)
Bases: *treeview_models.EmptyParameterValueModel*

An empty object parameter value model. Implements *items_to_add*.

items_to_add(*self*, *indexes*)
A dictionary of rows (int) to items (dict) to add to the db. Extend set of indexes as additional data is set.

class treeview_models.**EmptyRelationshipParameterValueModel**(*parent*)
Bases: *treeview_models.EmptyParameterValueModel*

An empty relationship parameter value model. Reimplements alsmot all methods from the super class EmptyParameterModel.

batch_set_data(*self*, *indexes*, *data*)
Batch set data for indexes. A little different from the base class implementation, since here we need to support creating relationships on the fly.

relationships_on_the_fly(*self*, *indexes*)
A dict of row (int) to relationship item (KeyedTuple), which can be either retrieved or added on the fly. Extend set of indexes as additional data is set.

add_relationships(*self*, *relationships_to_add*)
Add relationships to database on the fly and return them.

items_to_add(*self*, *indexes*, *relationships_on_the_fly*)
A dictionary of rows (int) to items (dict) to add to the db. Extend set of indexes as additional data is set.

class treeview_models.**EmptyParameterDefinitionModel**(*parent*)
Bases: *treeview_models.EmptyParameterModel*

An empty parameter definition model.

add_items_to_db(*self*, *items_to_add*)
Add parameter definitions to database.

class treeview_models.**EmptyObjectParameterDefinitionModel**(*parent*)
Bases: *treeview_models.EmptyParameterDefinitionModel*

An empty object parameter definition model.

items_to_add(*self*, *indexes*)
Return a dictionary of rows (int) to items (dict) to add to the db.

class treeview_models.**EmptyRelationshipParameterDefinitionModel**(*parent*)
Bases: *treeview_models.EmptyParameterDefinitionModel*

An empty relationship parameter definition model.

items_to_add(*self*, *indexes*)
Return a dictionary of rows (int) to items (dict) to add to the db. Extend set of indexes as additional data is set.

class treeview_models.**ObjectParameterModel**(*parent=None*)
Bases: *models.MinimalTableModel*

A model that concatenates several 'sub' object parameter models, one per object class.

flags(*self*, *index*)
Return flags for given index. Depending on the index's row we will land on a specific model. Models whose object class id is not selected are skipped.

data(*self*, *index*, *role=Qt.DisplayRole*)
Return data for given index and role. Depending on the index's row we will land on a specific model. Models whose object class id is not selected are skipped.

rowCount(*self*, *parent=QModelIndex()*)
Return the sum of rows in all models. Skip models whose object class id is not selected.

batch_set_data(*self*, *indexes*, *data*)
Batch set data for indexes. Distribute indexes and data among the different submodels and call batch_set_data on each of them.

insertRows(*self*, *row*, *count*, *parent=QModelIndex()*)
Find the right sub-model (or the empty model) and call insertRows on it.

removeRows(*self*, *row*, *count*, *parent=QModelIndex()*)
Find the right sub-models (or empty model) and call removeRows on them.

_handle_empty_rows_inserted(*self*, *parent*, *first*, *last*)

invalidate_filter(*self*)
Invalidate filter.

auto_filter_values(*self*, *column*)
Return values to populate the auto filter of given column. Each 'row' in the returned value consists of: 1) The 'checked' state, True if the value *hasn't* been filtered out 2) The value itself (an object name, a parameter name, a numerical value...) 3) A set of object class ids where the value is found.

set_filtered_out_values(*self*, *column*, *values*)
Set values that need to be filtered out.

clear_filtered_out_values(*self*)
Clear the set of values that need to be filtered out.

rename_object_classes(*self*, *db_map*, *object_classes*)
Rename object classes in model.

rename_parameter_tags(*self*, *db_map*, *parameter_tags*)
Rename parameter tags in model.

remove_object_classes(*self*, *db_map*, *object_classes*)
Remove object classes from model.

remove_parameter_tags(*self*, *db_map*, *parameter_tag_ids*)
Remove parameter tags from model.

_emit_data_changed_for_column(*self*, *column*)
Emits data changed for an entire column. Used by *rename_* and some *remove_* methods where it's too difficult to find out the exact rows that changed, especially because of filter status.

class treeview_models.**ObjectParameterValueModel**(*parent=None*)
Bases: *treeview_models.ObjectParameterModel*

A model that concatenates several 'sub' object parameter value models, one per object class.

reset_model(*self*, *main_data=None*)
Reset model data. Each sub-model is filled with parameter value data for a different object class.

update_filter(*self*)
Update filter.

rename_objects(*self*, *db_map*, *objects*)
Rename objects in model.

rename_parameter(*self*, *db_map*, *parameter*)
Rename single parameter in model.

remove_objects(*self*, *db_map*, *objects*)
Remove objects from model.

remove_parameters(*self*, *db_map*, *parameters*)
Remove parameters from model.

move_rows_to_sub_models(*self*, *rows*)
Move rows from empty row model to the a new sub_model. Called when the empty row model succesfully inserts new data in the db.

class treeview_models.**ObjectParameterDefinitionModel**(*parent=None*)
Bases: *treeview_models.ObjectParameterModel*

A model that concatenates several object parameter definition models (one per object class) vertically.

reset_model(*self*, *main_data=None*)
Reset model data. Each sub-model is filled with parameter definition data for a different object class.

update_filter(*self*)
Update filter.

move_rows_to_sub_models(*self*, *rows*)
Move rows from empty row model to a new sub_model. Called when the empty row model succesfully inserts new data in the db.

clear_parameter_value_lists(*self*, *db_map*, *value_list_ids*)
Clear parameter value_lists from model.

rename_parameter_value_lists(*self*, *db_map*, *value_lists*)
Rename parameter value_lists in model.

class treeview_models.**RelationshipParameterModel**(*parent=None*)
Bases: *models.MinimalTableModel*

A model that combines several relationship parameter models (one per relationship class), one on top of the other.

add_object_class_id_lists(*self*, *db_map*, *wide_relationship_class_list*)
Populate a dictionary of object class id lists per relationship class.

flags(*self*, *index*)
Return flags for given index. Depending on the index's row we will land on a specific model. Models whose relationship class id is not selected are skipped. Models whose object class id list doesn't intersect the selected ones are also skipped.

data(*self*, *index*, *role=Qt.DisplayRole*)
Return data for given index and role. Depending on the index's row we will land on a specific model. Models whose relationship class id is not selected are skipped. Models whose object class id list doesn't intersect the selected ones are also skipped.

rowCount(*self*, *parent=QModelIndex()*)
Return the sum of rows in all models. Models whose relationship class id is not selected are skipped. Models whose object class id list doesn't intersect the selected ones are also skipped.

batch_set_data(*self*, *indexes*, *data*)
Batch set data for indexes. Distribute indexes and data among the different submodels and call batch_set_data on each of them.

insertRows(*self*, *row*, *count*, *parent=QModelIndex()*)
Find the right sub-model (or the empty model) and call insertRows on it.

removeRows(*self*, *row*, *count*, *parent=QModelIndex()*)
Find the right sub-models (or empty model) and call removeRows on them.

_handle_empty_rows_inserted(*self*, *parent*, *first*, *last*)

invalidate_filter(*self*)
Invalidate filter.

auto_filter_values(*self*, *column*)
Return values to populate the auto filter of given column. Each 'row' in the returned value consists of: 1) The 'checked' state, True if the value *hasn't* been filtered out 2) The value itself (an object name, a parameter name, a numerical value...) 3) A set of relationship class ids where the value is found.

set_filtered_out_values(*self*, *column*, *values*)
Set values that need to be filtered out.

clear_filtered_out_values (*self*)
Clear the set of filtered out values.

rename_object_classes (*self*, *db_map*, *object_classes*)
Rename object classes in model.

rename_relationship_classes (*self*, *db_map*, *relationship_classes*)
Rename relationship classes in model.

rename_parameter_tags (*self*, *db_map*, *parameter_tags*)
Rename parameter tags in model.

remove_object_classes (*self*, *db_map*, *object_classes*)
Remove object classes from model.

remove_relationship_classes (*self*, *db_map*, *relationship_classes*)
Remove relationship classes from model.

remove_parameter_tags (*self*, *db_map*, *parameter_tag_ids*)
Remove parameter tags from model.

_emit_data_changed_for_column (*self*, *column*)
Emits data changed for an entire column. Used by *rename_* and some *remove_* methods where it's too difficult to find out the exact rows that changed, especially because of filter status.

class treeview_models.**RelationshipParameterValueModel** (*parent=None*)
Bases: *treeview_models.RelationshipParameterModel*

A model that combines several relationship parameter value models (one per relationship class), one on top of the other.

reset_model (*self*, *main_data=None*)
Reset model data. Each sub-model is filled with parameter value data for a different relationship class.

update_filter (*self*)
Update filter.

move_rows_to_sub_models (*self*, *rows*)
Move rows from empty row model to a new sub_model. Called when the empty row model succesfully inserts new data in the db.

rename_objects (*self*, *db_map*, *objects*)
Rename objects in model.

remove_objects (*self*, *db_map*, *objects*)
Remove objects from model.

remove_relationships (*self*, *db_map*, *relationships*)
Remove relationships from model.

rename_parameter (*self*, *db_map*, *parameter*)
Rename single parameter in model.

remove_parameters (*self*, *db_map*, *parameters*)
Remove parameters from model.

class treeview_models.**RelationshipParameterDefinitionModel** (*parent=None*)
Bases: *treeview_models.RelationshipParameterModel*

A model that combines several relationship parameter definition models (one per relationship class), one on top of the other.

reset_model (*self*, *main_data=None*)
Reset model data. Each sub-model is filled with parameter definition data for a different relationship class.

update_filter(*self*)
Update filter.

move_rows_to_sub_models(*self*, *rows*)
Move rows from empty row model to a new sub_model. Called when the empty row model succesfully inserts new data in the db.

clear_parameter_value_lists(*self*, *db_map*, *value_list_ids*)
Clear parameter value_lists from model.

rename_parameter_value_lists(*self*, *db_map*, *value_lists*)
Rename parameter value_lists in model.

class treeview_models.**ObjectParameterDefinitionFilterProxyModel**(*parent*, *parameter_definition_id_column*)
Bases: PySide2.QtCore.QSortFilterProxyModel

A filter proxy model for object parameter models.

update_filter(*self*, *parameter_definition_ids*)
Update filter.

set_filtered_out_values(*self*, *column*, *values*)
Set values that need to be filtered out.

clear_filtered_out_values(*self*)
Clear the filtered out values.

auto_filter_accepts_row(*self*, *source_row*, *source_parent*, *ignored_columns=None*)
Accept or reject row.

main_filter_accepts_row(*self*, *source_row*, *source_parent*)
Accept or reject row.

filterAcceptsRow(*self*, *source_row*, *source_parent*)
Accept or reject row.

batch_set_data(*self*, *indexes*, *data*)

class treeview_models.**ObjectParameterValueFilterProxyModel**(*parent*, *parameter_definition_id_column*, *object_id_column*, *db_column*)
Bases: *treeview_models.ObjectParameterDefinitionFilterProxyModel*

A filter proxy model for object parameter value models.

update_filter(*self*, *parameter_definition_ids*, *object_ids*)
Update filter.

main_filter_accepts_row(*self*, *source_row*, *source_parent*)
Accept or reject row.

class treeview_models.**RelationshipParameterDefinitionFilterProxyModel**(*parent*, *parameter_definition_id_column*)
Bases: PySide2.QtCore.QSortFilterProxyModel

A filter proxy model for relationship parameter definition models.

update_filter(*self*, *parameter_definition_ids*)
Update filter.

set_filtered_out_values (*self*, *column*, *values*)
Set values that need to be filtered out.

clear_filtered_out_values (*self*)
Clear the set of values that need to be filtered out.

auto_filter_accepts_row (*self*, *source_row*, *source_parent*, *ignored_columns=None*)
Accept or reject row.

main_filter_accepts_row (*self*, *source_row*, *source_parent*)
Accept or reject row.

filterAcceptsRow (*self*, *source_row*, *source_parent*)
Accept or reject row.

batch_set_data (*self*, *indexes*, *data*)

class treeview_models.**RelationshipParameterValueFilterProxyModel** (*parent*, *parameter_definition_id_column*, *object_id_list_column*, *db_column*)
Bases: *treeview_models.RelationshipParameterDefinitionFilterProxyModel*

A filter proxy model for relationship parameter value models.

update_filter (*self*, *parameter_definition_ids*, *object_ids*, *object_id_lists*)
Update filter.

main_filter_accepts_row (*self*, *source_row*, *source_parent*)
Accept or reject row.

class treeview_models.**TreeNode** (*parent*, *row*, *text=None*, *level=None*, *identifier=None*)
A helper class to use as the internalPointer of indexes in ParameterValueListModel.

Attributes parent (TreeNode): the parent node row (int): the row, needed by ParameterValueListModel.parent() text (str, NoneType): the text to show level (int, NoneType): the level in the tree id (int, NoneType): the id from the db table

class treeview_models.**ParameterValueListModel** (*parent*)
Bases: PySide2.QtCore.QAbstractItemModel

A class to display parameter value list data in a treeview.

build_tree (*self*)
Initialize the internal data structure of TreeNode instances.

index (*self*, *row*, *column*, *parent=QModelIndex()*)
Returns the index of the item in the model specified by the given row, column and parent index. Toplevel indexes get their pointer from the *_root_nodes* attribute; whereas inner indexes get their pointer from the *child_nodes* attribute of the parent node.

parent (*self*, *index*)
Returns the parent of the model item with the given index. Use the internal pointer to retrieve the parent node and use it to create the parent index.

rowCount (*self*, *parent=QModelIndex()*)
Returns the number of rows under the given parent. Get it from the lenght of the appropriate list.

columnCount (*self*, *parent=QModelIndex()*)
Returns the number of columns under the given parent. Always 1.

data (*self*, *index*, *role=Qt.DisplayRole*)
Returns the data stored under the given role for the item referred to by the index. Bold toplevel items. Get the DisplayRole from the *text* attribute of the internal pointer.

flags (*self*, *index*)
Returns the item flags for the given index.

setData (*self*, *index*, *value*, *role=Qt.EditRole*)
Sets the role data for the item at index to value. Returns True if successful; otherwise returns False. Basically just update the *text* attribute of the internal pointer.

appendRows (*self*, *count*, *parent=QModelIndex()*)
Append count rows into the model. Items in the new row will be children of the item represented by the parent model index.

_handle_data_changed (*self*, *top_left*, *bottom_right*, *roles=None*)
Called when data in the model changes.

append_empty_rows (*self*, *index*)
Append emtpy rows if index is the last children, so the user can continue editing the model.

items_to_add_and_update (*self*, *first*, *last*, *parent*)
Return list of items to add and update in the db.

batch_set_data (*self*, *indexes*, *values*)
Set edit role for indexes to values in batch.

removeRow (*self*, *row*, *parent=QModelIndex()*)
Remove row under parent, but never the last row (which is the empty one)

class treeview_models.**LazyLoadingArrayModel** (*parent*, *stride=256*)
Bases: *models.EmptyRowModel*

A model of array data, used by TreeViewForm.

parent
the parent widget

Type *JSONEditor*

stride
The number of elements to fetch

Type int

reset_model (*self*, *data*)
Store given array into the *_orig_data* attribute. Initialize first *_stride* rows of the model.

canFetchMore (*self*, *parent*)

fetchMore (*self*, *parent*)
Pop data from the _orig_data attribute and add it to the model.

all_data (*self*)
Return all data into a list.

16.20 tool_templates

Tool template classes.

authors

P. Savolainen (VTT), E. Rinne (VTT)

date 24.1.2018

16.20.1 Module Contents

class tool_templates.**ToolTemplate**(*toolbox*, *name*, *tooltype*, *path*, *includes*, *description=None*, *inputfiles=None*, *inputfiles_opt=None*, *outputfiles=None*, *cmdline_args=None*, *execute_in_work=True*)

Bases: *metaobject.MetaObject*

Super class for various tool templates.

toolbox
QMainWindow instance

Type ToolBoxUI

name
Name of the tool

Type str

description
Short description of the tool

Type str

path
Path to tool

Type str

includes
List of files belonging to the tool template (relative to 'path') # TODO: Change to src_files

Type str

inputfiles
List of required data files

Type list

inputfiles_opt
List of optional data files (wildcards may be used)

Type list, optional

outputfiles
List of output files (wildcards may be used)

Type list, optional

cmdline_args
Tool command line arguments (read from tool definition file)

Type str, optional

execute_in_work
Execute in work folder?

Type bool

set_return_code(*self*, *code*, *description*)
Set a return code and associated text description for the tool.

Parameters

- **code** (*int*) – Return code
- **description** (*str*) – Description

set_def_path(*self*, *path*)
Set definition file path for tool.

Parameters **path** (*str*) – Absolute path to the definition file.

get_def_path(*self*)
Returns tool definition file path.

static check_definition(*ui*, *data*)
Check that a tool template definition contains the required keys and that it is in correct format.

Parameters

- **ui** (ToolboxUI) – QMainWindow instance
- **data** (*dict*) – Tool template definition

Returns Dictionary or None if there was a problem in the tool definition.

class tool_templates.**GAMSTool**(*toolbox*, *name*, *tooltype*, *path*, *includes*, *description=None*, *inputfiles=None*, *inputfiles_opt=None*, *outputfiles=None*, *cmdline_args=None*, *execute_in_work=True*)
Bases: *tool_templates.ToolTemplate*

Class for GAMS tool templates.

name
GAMS Tool name

Type str

description
GAMS Tool description

Type str

path
Path to model main file

Type str

includes
List of files belonging to the tool (relative to 'path'). # TODO: Change to src_files

Type str

First file in the list is the main GAMS program.

inputfiles
List of required data files

Type list

inputfiles_opt
List of optional data files (wildcards may be used)

Type list, optional

outputfiles
List of output files (wildcards may be used)

Type list, optional

cmdline_args
GAMS tool command line arguments (read from tool definition file)

Type str, optional

__repr__(*self*)
Remove this if not necessary.

update_gams_options(*self*, *key*, *value*)
Update GAMS command line options. Only 'cerr and 'logoption' keywords supported.

Parameters

- **key** – Option name
- **value** – Option value

static load(*toolbox*, *path*, *data*)
Create a GAMSTool according to a tool definition.

Parameters

- **toolbox** (ToolboxUI) – QMainWindow instance
- **path** (*str*) – Base path to tool files
- **data** (*dict*) – Dictionary of tool definitions

Returns GAMSTool instance or None if there was a problem in the tool definition file.

class tool_templates.**JuliaTool**(*toolbox*, *name*, *tooltype*, *path*, *includes*, *description=None*, *inputfiles=None*, *inputfiles_opt=None*, *outputfiles=None*, *cmdline_args=None*, *execute_in_work=True*)
Bases: *tool_templates.ToolTemplate*

Class for Julia tool templates.

name
Julia Tool name

Type str

description
Julia Tool description

Type str

path
Path to model main file

Type str

includes
List of files belonging to the tool (relative to 'path'). # TODO: Change to src_files

Type str

First file in the list is the main Julia program.

inputfiles
List of required data files

Type list

inputfiles_opt
List of optional data files (wildcards may be used)

Type list, optional

outputfiles
List of output files (wildcards may be used)

Type list, optional

cmdline_args
Julia tool command line arguments (read from tool definition file)

Type str, optional

__repr__(*self*)
Remove this if not necessary.

update_julia_options(*self*, *key*, *value*)
Update Julia command line options.

Parameters

- **key** – Option name
- **value** – Option value

static load(*toolbox*, *path*, *data*)
Create a JuliaTool according to a tool definition.

Parameters

- **toolbox** (ToolboxUI) – QMainWindow instance
- **path** (*str*) – Base path to tool files
- **data** (*dict*) – Dictionary of tool definitions

Returns JuliaTool instance or None if there was a problem in the tool definition file.

class tool_templates.**PythonTool**(*toolbox*, *name*, *tooltype*, *path*, *includes*, *description=None*, *inputfiles=None*, *inputfiles_opt=None*, *outputfiles=None*, *cmdline_args=None*, *execute_in_work=True*)
Bases: *tool_templates.ToolTemplate*

Class for Python tool templates.

name
Python Tool name

Type str

description
Python Tool description

Type str

path
Path to model main file

Type str

includes
List of files belonging to the tool (relative to 'path'). # TODO: Change to src_files

Type str

First file in the list is the main Python program.

inputfiles
List of required data files

Type list

inputfiles_opt
List of optional data files (wildcards may be used)

Type list, optional

outputfiles
List of output files (wildcards may be used)

Type list, optional

cmdline_args
Python tool command line arguments (read from tool definition file)

Type str, optional

__repr__(*self*)
Remove this if not necessary.

update_python_options(*self*, *key*, *value*)
Update Python command line options.

Parameters

- **key** – Option name
- **value** – Option value

static load(*toolbox*, *path*, *data*)
Create a PythonTool according to a tool definition.

Parameters

- **toolbox** (ToolboxUI) – QMainWindow instance
- **path** (*str*) – Base path to tool files
- **data** (*dict*) – Dictionary of tool definitions

Returns PythonTool instance or None if there was a problem in the tool definition file.

class tool_templates.**ExecutableTool**(*toolbox*, *name*, *tooltype*, *path*, *includes*, *description=None*, *inputfiles=None*, *inputfiles_opt=None*, *outputfiles=None*, *cmdline_args=None*, *execute_in_work=True*)
Bases: *tool_templates.ToolTemplate*

Class for Executable tool templates.

name
Tool name

Type str

description
Tool description

Type str

path
Path to main script file

Type str

includes
List of files belonging to the tool (relative to 'path'). # TODO: Change to src_files

Type str

First file in the list is the main script file.

inputfiles
List of required data files

Type list

inputfiles_opt
List of optional data files (wildcards may be used)

Type list, optional

outputfiles
List of output files (wildcards may be used)

Type list, optional

cmdline_args
Tool command line arguments (read from tool definition file)

Type str, optional

__repr__(*self*)
Remove this if not necessary.

static load(*toolbox*, *path*, *data*)
Create an ExecutableTool according to a tool specification.

Parameters

- **toolbox** (ToolboxUI) – QMainWindow instance
- **path** (*str*) – Base path to tool files
- **data** (*dict*) – Tool specification

Returns ExecutableTool instance or None if there was a problem in the tool specification.

16.21 tool_instance

Contains ToolInstance class.

authors

P. Savolainen (VTT), E. Rinne (VTT)

date 1.2.2018

16.21.1 Module Contents

class tool_instance.**ToolInstance**(*tool_template*, *toolbox*, *tool_output_dir*, *project*, *execute_in_work*)
Bases: PySide2.QtCore.QObject

Class for Tool instances.

Parameters

- **tool_template** (ToolTemplate) – Tool for which this instance is created
- **toolbox** (ToolboxUI) – QMainWindow instance
- **tool_output_dir** (*str*) – Directory where results are saved

- **project** (SpineToolboxProject) – Current project
- **execute_in_work** (*bool*) – True executes instance in work dir, False executes in Tool template source dir

Class Variables: instance_finished_signal (Signal): Signal to emit when a Tool instance has finished processing

instance_finished_signal

_checkout
Copies Tool template files to work directory.

execute(*self*)
Starts executing Tool template instance in Julia Console, Python Console or in a sub-process.

julia_repl_tool_finished(*self*, *ret*)
Runs when Julia tool using Julia Console has finished processing.

Parameters **ret** (*int*) – Tool template process return value

julia_tool_finished(*self*, *ret*)
Runs when Julia tool from command line (without REPL) has finished processing.

Parameters **ret** (*int*) – Tool template process return value

python_console_tool_finished(*self*, *ret*)
Runs when Python Tool in Python Console has finished processing.

Parameters **ret** (*int*) – Tool template process return value

python_tool_finished(*self*, *ret*)
Runs when Python tool from command line has finished processing.

Parameters **ret** (*int*) – Tool template process return value

gams_tool_finished(*self*, *ret*)
Runs when GAMS tool has finished processing.

Parameters **ret** (*int*) – Tool template process return value

executable_tool_finished(*self*, *ret*)
Runs when an executable tool has finished processing.

Parameters **ret** (*int*) – Tool template process return value

handle_output_files(*self*, *ret*)
Creates a timestamped result directory for Tool template output files. Starts copying Tool template output files from work directory to result directory and print messages to Event Log depending on how the operation went.

Parameters **ret** (*int*) – Tool template process return value

terminate_instance(*self*)
Terminates Tool instance execution.

remove(*self*)
[Obsolete] Removes Tool instance files from work directory.

copy_output(*self*, *target_dir*)
Copies Tool template output files from work directory to given target directory.

Parameters **target_dir** (*str*) – Destination directory for Tool template output files

Returns Contains two lists. The first list contains paths to successfully copied files. The second list contains paths (or patterns) of Tool template output files that were not found.

Return type tuple

Raises `OSError` – If creating a directory fails.

make_work_output_dirs(*self*)
Makes sure that work directory has the necessary output directories for Tool output files. Checks only "outputfiles" list. Alternatively you can add directories to "inputfiles" list in the tool definition file.

Returns True for success, False otherwise.

Return type bool

Raises `OSError` – If creating an output directory to work fails.

16.22 data_store

Module for data store class.

authors

P. Savolainen (VTT), M. Marin (KTH)

date 18.12.2017

16.22.1 Module Contents

class data_store.**DataStore**(*toolbox*, *name*, *description*, *url*, *x*, *y*)
Bases: *project_item.ProjectItem*

Data Store class.

toolbox
QMainWindow instance

Type *ToolboxUI*

name
Object name

Type str

description
Object description

Type str

url
SQLAlchemy url

Type str or dict

x
Initial X coordinate of item icon

Type int

y
Initial Y coordinate of item icon

Type int

parse_url (*self*, *url*)
Return a complete url dictionary from the given dict or string

make_signal_handler_dict (*self*)
Returns a dictionary of all shared signals and their handlers. This is to enable simpler connecting and disconnecting.

activate (*self*)
Load url into selections and connect signals.

deactivate (*self*)
Disconnect signals.

set_url (*self*, *url*)
Set url attribute. Used by Tool when passing on results.

url (*self*)
Return the url attribute, for saving the project.

make_url (*self*, *log_errors=True*)
Return a sqlalchemy url from the current url attribute or None if not valid.

project (*self*)
Returns current project or None if no project open.

get_icon (*self*)
Returns the item representing this Data Store on the scene.

set_path_to_sqlite_file (*self*, *file_path*)
Set path to SQLite file.

open_sqlite_file (*self*, *checked=False*)
Open file browser where user can select the path to an SQLite file that they want to use.

load_url_into_selections (*self*)
Load url attribute into shared widget selections. Used when activating the item, and creating a new Spine db.

refresh_host (*self*, *host=''*)
Refresh host from selections.

refresh_port (*self*, *port=''*)
Refresh port from selections.

refresh_database (*self*, *database=''*)
Refresh database from selections.

refresh_username (*self*, *username=''*)
Refresh username from selections.

refresh_password (*self*, *password=''*)
Refresh password from selections.

refresh_dialect (*self*, *dialect=''*)

enable_no_dialect (*self*)
Adjust widget enabled status to default when no dialect is selected.

enable_mssql (*self*)
Adjust controls to mssql connection specification.

enable_sqlite(*self*)
Adjust controls to sqlite connection specification.

enable_common(*self*)
Adjust controls to 'common' connection specification.

check_dialect(*self*, *dialect*)
Check if selected dialect is supported. Offer to install DBAPI if not.

Returns True if dialect is supported, False if not.

install_dbapi_pip(*self*, *dbapi*)
Install DBAPI using pip.

install_dbapi_conda(*self*, *dbapi*)
Install DBAPI using conda. Fails if conda is not installed.

open_tree_view(*self*, *checked=False*)
Open url in tree view form.

do_open_tree_view(*self*, *db_map*)
Open url in tree view form.

tree_view_form_destroyed(*self*)
Notify that tree view form has been destroyed.

open_graph_view(*self*, *checked=False*)
Open url in graph view form.

do_open_graph_view(*self*, *db_map*)
Open url in graph view form.

graph_view_form_destroyed(*self*)
Notify that graph view form has been destroyed.

open_tabular_view(*self*, *checked=False*)
Open url in Data Store tabular view.

do_open_tabular_view(*self*, *db_map*, *database*)
Open url in tabular view form.

tabular_view_form_destroyed(*self*)

open_directory(*self*, *checked=False*)
Open file explorer in this Data Store's data directory.

data_files(*self*)
Return a list of files that are in this items data directory.

copy_url(*self*, *checked=False*)
Copy db url to clipboard.

create_new_spine_database(*self*, *checked=False*)
Create new (empty) Spine database.

do_create_new_spine_database(*self*, *url*, *for_spine_model*)
Separate method so 'busy_effect' don't overlay any message box.

update_name_label(*self*)
Update Data Store tab name label. Used only when renaming project items.

execute(*self*)
Executes this Data Store.

stop_execution(*self*)
Stops executing this Data Store.

16.23 qsubprocess

Module to handle running tools in a QProcess.

author

P. Savolainen (VTT)

date 1.2.2018

16.23.1 Module Contents

class qsubprocess.**QSubProcess**(*toolbox*, *program=None*, *args=None*, *silent=False*, *semisilent=False*)
Bases: PySide2.QtCore.QObject

Class to handle starting, running, and finishing PySide2 QProcesses.

subprocess_finished_signal

program(*self*)
Program getter method.

args(*self*)
Program argument getter method.

start_process(*self*, *workdir=None*)
Start the execution of a command in a QProcess.

Parameters **workdir** (*str*) – Script directory

wait_for_finished(*self*, *msecs=30000*)
Wait for subprocess to finish.

Returns True if process finished successfully, False otherwise

process_started(*self*)
Run when subprocess has started.

on_state_changed(*self*, *new_state*)
Runs when QProcess state changes.

Parameters **new_state** (*QProcess::ProcessState*) – Process state number

on_process_error(*self*, *process_error*)
Run if there is an error in the running QProcess.

Parameters **process_error** (*QProcess::ProcessError*) – Process error number

terminate_process(*self*)
Shutdown simulation in a QProcess.

process_finished(*self*, *exit_code*)
Run when subprocess has finished.

Parameters **exit_code** (*int*) – Return code from external program (only valid for normal exits)

on_ready_stdout (*self*)
Emit data from stdout.

on_ready_stderr (*self*)
Emit data from stderr.

16.24 metaobject

MetaObject class.

authors

E. Rinne (VTT), P. Savolainen (VTT)

date 18.12.2017

16.24.1 Module Contents

class metaobject.**MetaObject** (*name*, *description*)
Bases: PySide2.QtCore.QObject

Class for an object which has a name, type, and some description.

name
Object name

Type str

description
Object description

Type str

set_name (*self*, *new_name*)
Set object name and short name. Note: Check conflicts (e.g. name already exists) before calling this method.

Parameters **new_name** (*str*) – New (long) name for this object

set_description (*self*, *desc*)
Set object description.

Parameters **desc** (*str*) – Object description

16.25 tabularview_models

Spine Toolbox grid view

author

P. Vennström (VTT)

date 1.11.2018

16.25.1 Module Contents

class tabularview_models.**PivotModel**

rows

columns

clear_track_data(*self*)
clears data that is tracked

set_new_data(*self*, *data*, *index_names*, *index_type*, *rows=()*, *columns=()*, *frozen=()*, *frozen_value=()*, *index_entries=None*, *valid_index_values=None*, *tuple_index_entries=None*, *used_index_values=None*, *index_real_names=None*)
set the data of the model, index names and any additional indexes that don't have data, valid index values.

static _is_invalid_pivot(*rows*, *columns*, *frozen*, *frozen_value*, *index_names*)
checks if given pivot is valid for index_names, returns str with error message if invalid else None

_change_index_frozen(*self*)
Filters out data with index values in index_frozen

_index_key_getter(*self*, *names_of_index*)
creates a itemgetter that always returns tuples from list of index names

_get_unique_index_values(*self*, *index*, *filter_index*, *filter_value*)
Finds unique index values for index names in index filtered by index names in filter_index with values in filter_value

set_pivot(*self*, *rows*, *columns*, *frozen*, *frozen_value*)
Sets pivot for current data

set_frozen_value(*self*, *value*)
Sets the value of the frozen indexes

static _index_entries_without_data(*pivot_index*, *pivot_set*, *filter_index*, *filter_value*, *tuple_index_entries*)
find values in tuple_index_entries that are not present in pivot_set for index in pivot index filtered by filter_index and filter_value

get_pivoted_data(*self*, *row_mask*, *col_mask*)
gets data from current pivot with indexes in row_mask and col_mask

set_pivoted_data(*self*, *data*, *row_mask*, *col_mask*)
paste list of lists into current pivot, no change of indexes, row_mask list of indexes where to paste data rows in current pivot col_mask list of indexes where to paste data columns in current pivot

_add_index_value(*self*, *value*, *name*)

_delete_data(*self*, *key*)

_add_data(*self*, *key*, *value*)

_restore_data(*self*, *key*)

row(*self*, *row*)

column(*self*, *col*)

restore_pivoted_values(*self*, *indexes*)
Restores all values for given indexes

delete_pivoted_values(*self*, *indexes*)
Deletes values for given indexes

delete_tuple_index_values(*self*, *delete_tuples*)
deletes values from keys with combination of indexes given that match tuple_index_entries

delete_index_values(*self*, *delete_indexes*)
delete one ore more index value from data

_data_to_header(*self*, *data*, *start_index*, *index_values*, *index_names*, *mask*, *direction*)

paste_data(*self*, *row_start=0*, *row_header_data=None*, *col_start=0*, *col_header_data=None*, *data=None*, *row_mask=None*, *col_mask=None*)
Paste a list of list into current view of AbstractTable

edit_index(*self*, *new_index*, *index_mask*, *direction*)
Edits the index of either row or column

is_valid_index(*self*, *index*, *index_name*)
checks if if given index value is a valid value for given index

is_valid_key(*self*, *key*, *existing_keys*, *key_names*)
Checks if given key (combination of indexes) is valid

class tabularview_models.**PivotTableModel**(*parent=None*)
Bases: PySide2.QtCore.QAbstractTableModel

index_entries_changed

plot_x_column
Returns the index of the column designated as Y values for plotting or None.

set_data(*self*, *data*, *index_names*, *index_type*, *rows=()*, *columns=()*, *frozen=()*, *frozen_value=()*, *index_entries=None*, *valid_index_values=None*, *tuple_index_entries=None*, *used_index_values=None*, *index_real_names=None*)

set_pivot(*self*, *rows*, *columns*, *frozen*, *frozen_value*)

set_frozen_value(*self*, *frozen_value*)

delete_values(*self*, *indexes*)

delete_index_values(*self*, *keys_dict*)

delete_tuple_index_values(*self*, *tuple_key_dict*)

restore_values(*self*, *indexes*)

get_key(*self*, *index*)

get_col_key(*self*, *column*)

paste_data(*self*, *index*, *data*, *row_mask*, *col_mask*)
paste data into pivot model

_indexes_to_pivot_index(*self*, *indexes*)

_update_header_data(*self*)
updates the top left corner 'header' data

first_data_row(*self*)
Returns the row index to the first data row.

dataRowCount(*self*)
number of rows that contains actual data

dataColumnCount(*self*)
number of columns that contains actual data

rowCount(*self*, *parent=QModelIndex()*)
Number of rows in table, number of header rows + datarows + 1 empty row

columnCount(*self*, *parent=QModelIndex()*)
Number of columns in table, number of header columns + datacolumns + 1 empty columns

flags(*self*, *index*)
Roles for data

index_in_top_left(*self*, *index*)
check if index is in top left corner, where pivot names are displayed

index_in_data(*self*, *index*)
check if index is in data area

index_in_column_headers(*self*, *index*)
check if index is in column headers (horizontal) area

index_in_row_headers(*self*, *index*)
check if index is in row headers (vertical) area

set_plot_x_column(*self*, *column*, *is_x*)
Sets or clears the Y flag on a column

set_index_key(*self*, *index*, *value*, *direction*)
edits/sets a index value in a index in row/column

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

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

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

data_color(*self*, *index*)

class tabularview_models.**PivotTableSortFilterProxy**(*parent=None*)
Bases: PySide2.QtCore.QSortFilterProxyModel

set_filter(*self*, *index_name*, *filter_value*)

clear_filter(*self*)

accept_index(*self*, *index*, *index_names*)

delete_values(*self*, *delete_indexes*)

restore_values(*self*, *indexes*)

paste_data(*self*, *index*, *data*)

filterAcceptsRow(*self*, *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. All the rules and subrules need to pass.

filterAcceptsColumn(*self*, *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.

class tabularview_models.**FilterCheckboxListModel**(*parent=None*, *show_empty=True*)
Bases: PySide2.QtCore.QAbstractListModel

reset_selection(*self*)

_select_all_clicked(*self*)

_is_all_selected(*self*)

rowCount (*self*, *parent=QModelIndex()*)

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

click_index (*self*, *index*)

set_list (*self*, *data*, *all_selected=True*)

add_item (*self*, *items*, *selected=True*)

set_selected (*self*, *selected*, *select_empty=None*)

get_selected (*self*)

get_not_selected (*self*)

set_filter (*self*, *search_for*)

apply_filter (*self*)

_remove_and_add_filtered (*self*)

_remove_and_replace_filtered (*self*)

remove_filter (*self*)

remove_items (*self*, *items*)

16.26 data_interface

Contains DataInterface class.

authors

P. Savolainen (VTT)

date 10.6.2019

16.26.1 Module Contents

class data_interface.**DataInterface** (*toolbox*, *name*, *description*, *filepath*, *settings*, *x*, *y*)

Bases: *project_item.ProjectItem*

DataInterface class.

toolbox
QMainWindow instance

Type *ToolboxUI*

name
Project item name

Type str

description
Project item description

Type str

filepath
Path to file

Type str

settings
dict with mapping settings

Type dict

x
Initial icon scene X coordinate

Type int

y
Initial icon scene Y coordinate

Type int

data_interface_refresh_signal

_handle_file_model_item_changed(*self*, *item*)

make_signal_handler_dict(*self*)
Returns a dictionary of all shared signals and their handlers. This is to enable simpler connecting and disconnecting.

activate(*self*)
Restores selections and connects signals.

deactivate(*self*)
Saves selections and disconnects signals.

restore_selections(*self*)
Restores selections into shared widgets when this project item is selected.

save_selections(*self*)
Saves selections in shared widgets for this project item into instance variables.

get_icon(*self*)
Returns the graphics item representing this data interface on scene.

update_name_label(*self*)
Update Data Interface tab name label. Used only when renaming project items.

open_directory(*self*, *checked=False*)
Opens file explorer in Data Interface directory.

_handle_import_editor_clicked(*self*, *checked=False*)
Opens Import editor for the file selected in list view.

_handle_files_double_clicked(*self*, *index*)
Opens Import editor for the double clicked index.

open_import_editor(*self*, *index*)
Opens Import editor for the given index.

get_connector(*self*, *importee*)
Shows a QDialog to select a connector for the given source file. Mimics similar routine in *spine_io.widgets.import_widget.ImportDialog*

select_connector_type(*self*, *index*)
Opens dialog to select connector type for the given index.

_connection_failed(*self*, *msg*, *importee*)

save_settings(*self*, *settings*, *importee*)

_preview_destroyed(*self*, *importee*)

update_file_model (*self*, *items*)
Add given list of items to the file model. If None or an empty list given, the model is cleared.

refresh (*self*)
Update the list of files that this item is viewing.

execute (*self*)
Executes this Data Interface.

stop_execution (*self*)
Stops executing this Data Interface.

16.27 view

Module for view class.

authors

P. Savolainen (VTT), M. Marin (KHT), J. Olauson (KTH)

date 14.07.2018

16.27.1 Module Contents

class view.**View** (*toolbox*, *name*, *description*, *x*, *y*)
Bases: *project_item.ProjectItem*

View class.

toolbox
QMainWindow instance

Type *ToolboxUI*

name
Object name

Type str

description
Object description

Type str

x
Initial X coordinate of item icon

Type int

y
Initial Y coordinate of item icon

Type int

view_refresh_signal

make_signal_handler_dict (*self*)
Returns a dictionary of all shared signals and their handlers. This is to enable simpler connecting and disconnecting.

activate (*self*)
Restore selections and connect signals.

deactivate (*self*)
Save selections and disconnect signals.

restore_selections (*self*)
Restore selections into shared widgets when this project item is selected.

save_selections (*self*)
Save selections in shared widgets for this project item into instance variables.

get_icon (*self*)
Returns the item representing this Data Store on the scene.

references (*self*)
Returns a list of url strings that are in this item as references.

find_input_items (*self*)
Find input project items (only Data Stores now) that are connected to this View.

Returns List of Data Store items.

refresh (*self*)
Update the list of references that this item is viewing.

open_graph_view_btn_clicked (*self*, *checked=False*)
Slot for handling the signal emitted by clicking on 'Graph view' button.

open_tabular_view_btn_clicked (*self*, *checked=False*)
Slot for handling the signal emitted by clicking on 'Tabular view' button.

open_tree_view_btn_clicked (*self*, *checked=False*)
Slot for handling the signal emitted by clicking on 'Tree view' button.

_open_view (*self*, *view_store*, *supports_multiple_databases*)
Opens references in a view window.

Parameters

- **view_store** (*dict*) – a dictionary where to store the view window
- **supports_multiple_databases** (*bool*) – True if the view supports more than one database

close_all_views (*self*)
Closes all view windows.

populate_reference_list (*self*, *items*)
Add given list of items to the reference model. If None or an empty list given, the model is cleared.

update_name_label (*self*)
Update View tab name label. Used only when renaming project items.

open_directory (*self*, *checked=False*)
Open file explorer in View data directory.

execute (*self*)
Executes this View.

stop_execution (*self*)
Stops executing this View.

_selected_indexes (*self*)
Returns selected indexes.

_database_maps(*self*, *indexes*)
Returns database maps and database paths for given indexes.

static _restore_existing_view_window(*view_id*, *view_store*)
Restores an existing view window and returns True if the operation was successful.

_make_view_window(*self*, *view_store*, *db_maps*, *databases*)

16.28 time_pattern_model

A model for time patterns, used by the parameter value editors.

authors

A. Soininen (VTT)

date 4.7.2019

16.28.1 Module Contents

class time_pattern_model.**TimePatternModel**(*value*)
Bases: *indexed_value_table_model.IndexedValueTableModel*

A model for time pattern type parameter values.

value
a time pattern value

Type TimePattern

flags(*self*, *index*)
Returns flags at index.

insertRows(*self*, *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

removeRows(*self*, *row*, *count*, *parent=QModelIndex()*)
Removes time period - value pairs from the pattern.

row
an index where to remove the data
Type int

count
number of time period - value pairs to remove
Type int

parent
an index to a parent model

Type QModelIndex

Returns True if the operation was successful

setData (*self*, *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.

index
an index to the model
Type QModelIndex

value
a new time period or value
Type str, float

role
a role
Type int

Returns True if the operation was successful

batch_set_data (*self*, *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

16.29 excel_import_export

Functions to import and export from excel to spine database.

author

P. Vennström (VTT)

date 21.8.2018

16.29.1 Module Contents

excel_import_export.**SheetData**

excel_import_export.**import_xlsx_to_db** (*db*, *filepath*)
reads excel file in 'filepath' and insert into database in mapping 'db'. Returns two list, one with succesful writes to database, one with errors when trying to write to database.

Parameters

- **db** (*spinedb_api.DatabaseMapping*) – database mapping for database to write to
- **filepath** (*str*) – str with filepath to excel file to read from

Returns (Int, List) Returns number of inserted items and a list of error information on all failed writes

excel_import_export.**get_objects_and_parameters**(*db*)
Exports all object data from spine database into unstacked list of lists

Parameters **db** (*spinedb_api.DatabaseMapping*) – database mapping for database

Returns (List, List) First list contains parameter data, second one json data

excel_import_export.**get_relationships_and_parameters**(*db*)
Exports all relationship data from spine database into unstacked list of lists

Parameters **db** (*spinedb_api.DatabaseMapping*) – database mapping for database

Returns (List, List) First list contains parameter data, second one json data

excel_import_export.**unstack_list_of_tuples**(*data*, *headers*, *key_cols*, *value_name_col*, *value_col*)
Unstacks list of lists or list of tuples and creates a list of namedtuples whit unstacked data (pivoted data)

Parameters

- **data** (*List[List]*) – List of lists with data to unstack
- **headers** (*List[str]*) – List of header names for data
- **key_cols** (*List[Int]*) – List of index for column that are keys, columns to not unstack
- **value_name_col** (*Int*) – index to column containing name of data to unstack
- **value_col** (*Int*) – index to column containing value to value_name_col

Returns List of list with headers in headers list (List): List of header names for each item in inner list

Return type (List[List])

excel_import_export.**stack_list_of_tuples**(*data*, *headers*, *key_cols*, *value_cols*)
Stacks list of lists or list of tuples and creates a list of namedtuples with stacked data (unpivoted data)

Parameters

- **data** (*List[List]*) – List of lists with data to unstack
- **headers** (*List[str]*) – List of header names for data
- **key_cols** (*List[Int]*) – List of index for columns that are keys
- **value_cols** (*List[Int]*) – List of index for columns containing values to stack

Returns List of namedtuples whit fields given by headers and 'parameter' and 'value' which contains stacked values

Return type (List[namedtuple])

excel_import_export.**unpack_json_parameters**(*data*, *json_index*)

excel_import_export.**pack_json_parameters**(*data*, *key_cols*, *value_col*, *index_col=None*)

excel_import_export.**get_unstacked_relationships**(*db*)
Gets all data for relationships in a unstacked list of list

Parameters **db** (*spinedb_api.DatabaseMapping*) – database mapping for database

Returns Two list of data for relationship, one with parameter values and the second one with json values

Return type (List, List)

excel_import_export.**get_unstacked_objects**(*db*)
Gets all data for objects in a unstacked list of list

Parameters **db** (*spinedb_api.DatabaseMapping*) – database mapping for database

Returns Two list of data for objects, one with parameter values and the second one with json values

Return type (List, List)

excel_import_export.**write_relationships_to_xlsx**(*wb*, *relationship_data*)
Writes Classes, parameter and parameter values for relationships. Writes one sheet per relationship class.

Parameters

- **wb** (*openpyxl.Workbook*) – excel workbook to write too.
- **relationship_data** (*List[List]*) – List of lists containing relationship
- **give by function get_unstacked_relationships** (*data*) –

excel_import_export.**write_json_array_to_xlsx**(*wb*, *data*, *sheet_type*)
Writes json array data for object classes and relationship classes. Writes one sheet per relationship/object class.

Parameters

- **wb** (*openpyxl.Workbook*) – excel workbook to write too.
- **data** (*List[List]*) – List of lists containing json data give by function
- **and get_unstacked_relationships** (*get_unstacked_objects*) –
- **sheet_type** (*str*) – str with value "relationship" or "object" telling if data is for a relationship or object

excel_import_export.**write_TimeSeries_to_xlsx**(*wb*, *data*, *sheet_type*, *data_type*)
Writes spinedb_api TimeSeries data for object classes and relationship classes. Writes one sheet per relationship/object class.

Parameters

- **wb** (*openpyxl.Workbook*) – excel workbook to write too.
- **data** (*List[List]*) – List of lists containing json data give by function
- **and get_unstacked_relationships** (*get_unstacked_objects*) –
- **sheet_type** (*str*) – str with value "relationship" or "object" telling if data is for a relationship or object

excel_import_export.**write_objects_to_xlsx**(*wb*, *object_data*)
Writes Classes, parameter and parameter values for objects. Writes one sheet per relationship/object class.

Parameters

- **wb** (*openpyxl.Workbook*) – excel workbook to write too.
- **object_data** (*List[List]*) – List of lists containing relationship data give by function get_unstacked_objects

excel_import_export.**export_spine_database_to_xlsx**(*db*, *filepath*)
Writes all data in a spine database into an excel file.

Parameters

- **db** (*spinedb_api.DatabaseMapping*) – database mapping for database.
- **filepath** (*str*) – str with filepath to save excel file to.

excel_import_export.**read_spine_xlsx**(*filepath*)
reads all data from a excel file where the sheets are in valid spine data format

Parameters **filepath** (*str*) – str with filepath to excel file to read from.

excel_import_export.**merge_spine_xlsx_data**(*data*)
Merge data from different sheets with same object class or relationship class.

Parameters **data** (*List(SheetData)*) – list of SheetData

Returns List of SheetData with only one relationship/object class per item

Return type (List[SheetData])

excel_import_export.**validate_sheet**(*ws*)
Checks if supplied sheet is a valid import sheet for spine.

Parameters **ws** (*openpyxl.workbook.worksheet*) – worksheet to validate

Returns True if sheet is valid, False otherwise

Return type (bool)

excel_import_export.**read_json_sheet**(*ws*, *sheet_type*)
Reads a sheet containg json array data for objects and relationships

Parameters

- **ws** (*openpyxl.workbook.worksheet*) – worksheet to read from
- **sheet_type** (*str*) – str with value "relationship" or "object" telling if sheet is a relationship or object sheet

Returns (List[SheetData])

excel_import_export.**read_TimeSeries_sheet**(*ws*, *sheet_type*)
Reads a sheet containg json array data for objects and relationships

Parameters

- **ws** (*openpyxl.workbook.worksheet*) – worksheet to read from
- **sheet_type** (*str*) – str with value "relationship" or "object" telling if sheet is a relationship or object sheet

Returns (List[SheetData])

excel_import_export.**read_parameter_sheet**(*ws*)
Reads a sheet containg parameter data for objects and relationships

Parameters **ws** (*openpyxl.workbook.worksheet*) – worksheet to read from

Returns (List[SheetData])

excel_import_export.**read_2d**(*ws*, *start_row=1*, *end_row=1*, *start_col=1*, *end_col=1*)
Reads a 2d area from worksheet into a list of lists where each line is the inner list.

Parameters

- **ws** (*openpyxl.workbook.worksheet*) – Worksheet to look in
- **start_row** (*Integer*) – start row to read, 1-indexed (as excel)
- **end_row** (*Integer*) – row to read to, 1-indexed (as excel)
- **start_col** (*Integer*) – start column to read, 1-indexed (as excel)
- **end_col** (*Integer*) – end column to read to, 1-indexed (as excel)

Returns (List) List of all lines read.

excel_import_export.**max_col_in_row**(*ws*, *row=1*)
Finds max col index for given row. If no data exists on row, returns 1.

Parameters

- **ws** (*openpyxl.workbook.worksheet*) – Worksheet to look in
- **row** (*Integer*) – index for row to search, 1-indexed (as excel)

Returns (Integer) column index of last cell with value.

16.30 spine_io

Init file for spine_io package. Intentionally empty.

author

P. Vennström (VTT)

date 1.6.2019

16.30.1 Subpackages

spine_io.importers

Intentionally empty.

author

P. Vennström (VTT)

date 1.6.2019

Submodules

spine_io.importers.csv_reader

Contains CSVConnector class and a help function.

author

P. Vennström (VTT)

date 1.6.2019

Module Contents

spine_io.importers.csv_reader.**select_csv_file**(*parent=None*)
Launches QFileDialog with no filter

class spine_io.importers.csv_reader.**CSVConnector**
Bases: *spine_io.io_api.SourceConnection*

Template class to read data from another QThread

DISPLAY_NAME = Text/CSV

OPTIONS

SELECT_SOURCE_UI

connect_to_source(*self*, *source*)
saves filepath

Parameters **{str} -- filepath**(*source*) –

disconnect(*self*)
Disconnect from connected source.

get_tables(*self*)
Method that should return a list of table names, list(str)

Raises NotImplementedError – [description]

static parse_options(*options*)
Parses options dict to dialect and quotechar options for csv.reader

Parameters **{dict} -- dict with options**(*options*) – "delimiter": file delimiter "quotechar": file quotechar "has_header": if first row should be treated as a header "skip": how many rows should be skipped

Returns

tuple(dict, bool, integer) – tuple dialect for csv.reader, quotechar for csv.reader and number of rows to skip

file_iterator(*self*, *options*, *max_rows*)
creates an iterator that reads max_rows number of rows from text file

Parameters

- **{dict} -- dict with options**(*options*) –
- **{integer} -- max number of rows to read, if -1 then read all rows**(*max_rows*) –

Returns iterator – iterator of csv file

get_data_iterator(*self*, *table*, *options*, *max_rows=-1*)
Creates a iterator for the file in self.filename

Parameters

- **{string} -- ignored, used in abstract IOWorker class**(*table*) –
- **{dict} -- dict with options**(*options*) –

Keyword Arguments **{int} -- how many rows of data to read, if -1 read all rows (default**(*max_rows*) – {-1})

Returns [type] – [description]

spine_io.importers.excel_reader

Contains ExcelConnector class and a help function.

author

P. Vennström (VTT)

date 1.6.2019

Module Contents

spine_io.importers.excel_reader.**select_excel_file**(*parent=None*)
Launches QFileDialog with .xlsx and friends filter

class spine_io.importers.excel_reader.**ExcelConnector**
Bases: *spine_io.io_api.SourceConnection*

Template class to read data from another QThread

DISPLAY_NAME = Excel

OPTIONS

SELECT_SOURCE_UI

connect_to_source(*self*, *source*)
saves filepath

Parameters **{str} -- filepath**(*source*) –

disconnect(*self*)
Disconnect from connected source.

get_tables(*self*)
Method that should return a list of table names, list(str)

Raises NotImplementedError – [description]

get_data_iterator(*self*, *table*, *options*, *max_rows=-1*)
Return data read from data source table in table. If max_rows is specified only that number of rows.

spine_io.importers.excel_reader.**create_mapping_from_sheet**(*worksheet*)
Checks if sheet is a valid spine excel template, if so creates a mapping object for each sheet.

spine_io.importers.gdx_connector

Contains GDXConnector class and a help function.

author

P. Vennström (VTT)

date 1.6.2019

Module Contents

spine_io.importers.gdx_connector.**IMPORT_ERROR =**

class spine_io.importers.gdx_connector.**GamsDataType**
Bases: enum.Enum

Set

Parameter

Variable

Equation

Alias

spine_io.importers.gdx_connector.**select_gdx_file**(*parent=None*)
Launches QFileDialog with .gdx filter

class spine_io.importers.gdx_connector.**GdxConnector**
Bases: *spine_io.io_api.SourceConnection*

Template class to read data from another QThread

DISPLAY_NAME = Gdx

OPTIONS

SELECT_SOURCE_UI

__exit__(*self*, *exc_type*, *exc_value*, *traceback*)

__del__(*self*)

connect_to_source(*self*, *source*)
saves filepath

Parameters **{str} -- filepath**(*source*) –

disconnect(*self*)
Disconnect from connected source.

get_tables(*self*)
Method that should return a list of table names, list(str)

Raises NotImplementedError – [description]

get_data_iterator(*self*, *table*, *options*, *max_rows=-1*)
Creates a iterator for the file in self.filename

Parameters

- **{string} -- ignored, used in abstract IOWorker class**(*table*) –
- **{dict} -- dict with options**(*options*) –

Keyword Arguments **{int} -- how many rows of data to read, if -1 read all rows (default**(*max_rows*) – {-1})

Returns [type] – [description]

spine_io.importers.odbc_reader

Contains ODBCConnector class.

author

P. Vennström (VTT)

date 1.6.2019

Module Contents

class spine_io.importers.odbc_reader.**ODBCConnector**
Bases: *spine_io.io_api.SourceConnection*

HAS_TABLES = True

DISPLAY_NAME = ODBC

tables
Tables.

connect_to_source (*self*, *source*)
TODO: Needs implementation

disconnect (*self*)
TODO: Needs implementation

get_tables (*self*)
TODO: Needs implementation

get_data_iterator (*self*, *table*, *options*, *max_rows=-1*)
TODO: Needs implementation

_new_options (*self*)

set_table (*self*, *table*)

source_selector (*self*, *parent=None*)

read_data (*self*, *table*, *max_rows=100*)
Return data read from data source table in table. If max_rows is specified only that number of rows.

preview_data (*self*, *table*)

option_widget (*self*)
Return a QWidget with options for reading data from a table in source.

spine_io.importers.sqlalchemy_connector

Contains SqlAlchemyConnector class and a help function.

author

P. Vennström (VTT)

date 1.6.2019

Module Contents

spine_io.importers.sqlalchemy_connector.**select_sa_conn_string** (*parent=None*)
Launches QInputDialog for entering connection string

class spine_io.importers.sqlalchemy_connector.**SqlAlchemyConnector**
Bases: *spine_io.io_api.SourceConnection*

Template class to read data from another QThread.

DISPLAY_NAME = SqlAlchemy

OPTIONS

SELECT_SOURCE_UI

connect_to_source (*self*, *source*)
saves filepath

Parameters **{str} -- filepath** (*source*) –

disconnect (*self*)
Disconnect from connected source.

get_tables(*self*)
Method that should return a list of table names, list(str)

Raises NotImplementedError – [description]

get_data_iterator(*self*, *table*, *options*, *max_rows=-1*)
Creates a iterator for the file in self.filename

Parameters

- **{string} -- table name**(*table*) –
- **{dict} -- dict with options, not used**(*options*) –

Keyword Arguments **{int} -- how many rows of data to read, if -1 read all rows (default**(*max_rows*) – {-1})

Returns [type] – [description]

16.30.2 Submodules

spine_io.connection_manager

Contains ConnectionManager class.

author

P. Vennström (VTT)

date 1.6.2019

Module Contents

class spine_io.connection_manager.**ConnectionManager**(*connection*, *parent=None*)
Bases: PySide2.QtCore.QObject

Class to manage data connections in another thread.

Parameters **connection**(*class*) – A class derived from *SourceConnection*, e.g. *CSVConnector*

startTableGet

startDataGet

startMappedDataGet

connectionFailed

connectionReady

closeConnection

error

fetchingData

dataReady

tablesReady

mappedDataReady

is_connected

table_options

source

source_type

set_table(*self*, *table*)
Sets the current table of the data source.

Parameters **{str} -- str with table name**(*table*) –

request_tables(*self*)
Get tables tables from source, emits two singals, fetchingData: ConnectionManager is busy waiting for data startTableGet: a signal that the worker in another thread is listening to know when to run get a list of table names.

request_data(*self*, *table=None*, *max_rows=-1*)
Request data from emits dataReady to with data

Keyword Arguments

- **{str} -- which table to get data from (default**(*table*) – {None})
- **{int} -- how many rows to read (default**(*max_rows*) – {-1})

request_mapped_data(*self*, *table_mappings*, *max_rows=-1*)
Get mapped data from csv file

Parameters **{dict} -- dict with filename as key and a list of mappings as value**(*table_mappings*) –

Keyword Arguments **{int} -- number of rows to read, if -1 read all rows (default**(*max_rows*) – {-1})

connection_ui(*self*)
launches a modal ui that prompts the user to select source.

ex: fileselect if source is a file.

init_connection(*self*)
Creates a Worker and a new thread to read source data. If there is an existing thread close that one.

_handle_connection_ready(*self*)

_handle_tables_ready(*self*, *table_options*)

_new_options(*self*)

set_table_options(*self*, *options*)
Sets connection manager options for current connector

Parameters **{dict} -- Dict with option settings**(*options*) –

option_widget(*self*)
Return a Qwidget with options for reading data from a table in source

close_connection(*self*)
Close and delete thread and worker

class spine_io.connection_manager.**ConnectionWorker**(*source*, *connection*, *parent=None*)

Bases: PySide2.QtCore.QObject

A class for delegating SourceConnection operations to another QThread.

Parameters

- **source** (*str*) – path of the source file
- **connection** (*class*) – A class derived from *SourceConnection* for connecting to the source file

connectionFailed

error

connectionReady

tablesReady

dataReady

mappedDataReady

init_connection(*self*)
Connect to data source

tables(*self*)

data(*self*, *table*, *options*, *max_rows*)

mapped_data(*self*, *table_mappings*, *options*, *max_rows*)

disconnect(*self*)

spine_io.io_api

Contains a class template for a data source connector used in import ui.

author

P. Vennström (VTT)

date 1.6.2019

Module Contents

class spine_io.io_api.**SourceConnection**
Template class to read data from another QThread

DISPLAY_NAME = unnamed source

OPTIONS

SELECT_SOURCE_UI

connect_to_source(*self*, *source*)
Connects to source, ex: connecting to a database where source is a connection string.

Parameters **{} -- object with information on source to be connected to, ex** (*source*) – filepath string for a csv connection

disconnect(*self*)
Disconnect from connected source.

get_tables(*self*)
Method that should return a list of table names, list(str)

Raises NotImplementedError – [description]

get_data_iterator(*self*, *table*, *options*, *max_rows=-1*)
Function that should return a data iterator, data header and number of columns.

get_data(*self*, *table*, *options*, *max_rows=-1*)
Return data read from data source table in table. If max_rows is specified only that number of rows.

get_mapped_data(*self*, *tables_mappings*, *options*, *max_rows=-1*)
Reads all mappings in dict tables_mappings, where key is name of table and value is the mappings for that table. emits mapped data when ready.

spine_io.io_models

Classes for handling models in PySide2's model/view framework.

author

P. Vennström (VTT)

date 1.6.2019

Module Contents

spine_io.io_models.**_DISPLAY_TYPE_TO_TYPE**

spine_io.io_models.**_TYPE_TO_DISPLAY_TYPE**

class spine_io.io_models.**MappingPreviewModel**(*parent=None*)
Bases: *models.MinimalTableModel*

A model for highlighting columns, rows, and so on, depending on Mapping specification. Used by ImportPreviewWidget.

set_mapping(*self*, *mapping*)
Set mapping to display colors from

Parameters **{MappingSpecModel} -- mapping model**(*mapping*) –

update_colors(*self*)

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

data_color(*self*, *index*)
returns background color for index depending on mapping

Parameters **{PySide2.QtCore.QModelIndex} -- index**(*index*) –

Returns [QColor] – QColor of index

index_in_mapping(*self*, *mapping*, *index*)
Checks if index is in mapping

Parameters

- **{Mapping} -- mapping**(*mapping*) –
- **{QModelIndex} -- index**(*index*) –

Returns [bool] – returns True if mapping is in index

mapping_column_ref_int_list(*self*)
Returns a list of column indexes that are not pivoted

Returns [List[int]] – list of ints

class spine_io.io_models.**MappingSpecModel**(*model*, *parent=None*)
Bases: PySide2.QtCore.QAbstractTableModel

A model to hold a Mapping specification.

map_type

dimension

import_objects

parameter_type

is_pivoted

set_import_objects(*self*, *flag*)

set_mapping(*self*, *mapping*)

set_dimension(*self*, *dim*)

change_model_class(*self*, *new_class*)
Change model between Relationship and Object class

change_parameter_type(*self*, *new_type*)
Change parameter type

update_display_table(*self*)

get_map_type_display(*self*, *mapping*, *name*)

get_map_value_display(*self*, *mapping*, *name*)

get_map_append_display(*self*, *mapping*, *name*)

get_map_prepend_display(*self*, *mapping*, *name*)

data(*self*, *index*, *role*)

rowCount(*self*, *index=None*)

columnCount(*self*, *index=None*)

headerData(*self*, *section*, *orientation*, *role*)

flags(*self*, *index*)

setData(*self*, *index*, *value*, *role*)

set_type(*self*, *name*, *value*)

set_value(*self*, *name*, *value*)

set_append_str(*self*, *name*, *value*)

set_prepend_str(*self*, *name*, *value*)

get_mapping_from_name(*self*, *name*)

set_mapping_from_name(*self*, *name*, *mapping*)

set_skip_columns(*self*, *columns=None*)

class spine_io.io_models.**MappingListModel**(*mapping_list*, *parent=None*)
Bases: PySide2.QtCore.QAbstractListModel

A model to hold a list of Mappings.

set_model(*self*, *model*)

get_mappings(*self*)

rowCount(*self*, *index=None*)

data_mapping(*self*, *index*)

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

add_mapping(*self*)

remove_mapping(*self*, *row*)

16.31 widgets

Init file for widgets package. Intentionally empty.

author

P. Savolainen (VTT)

date 3.1.2018

16.31.1 Submodules

widgets.about_widget

A widget for presenting basic information about the application.

author

P. Savolainen (VTT)

date 14.12.2017

Module Contents

class widgets.about_widget.**AboutWidget**(*toolbox*, *version*)
Bases: PySide2.QtWidgets.QWidget

About widget class.

toolbox
QMainWindow instance

Type *ToolboxUI*

version
Application version number

Type str

calc_pos(*self*)
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(*self*)
Add license to QTextBrowser.

keyPressEvent(*self*, *e*)
Close form when Escape, Enter, Return, or Space bar keys are pressed.

Parameters **e** (*QKeyEvent*) – Received key press event.

closeEvent(*self*, *event=None*)
Handle close window.

Parameters **event** (*QEvent*) – Closing event if 'X' is clicked.

mousePressEvent(*self*, *e*)
Save mouse position at the start of dragging.

Parameters **e** (*QMouseEvent*) – Mouse event

mouseReleaseEvent(*self*, *e*)
Save mouse position at the end of dragging.

Parameters **e** (*QMouseEvent*) – Mouse event

mouseMoveEvent(*self*, *e*)
Moves the window when mouse button is pressed and mouse cursor is moved.

Parameters **e** (*QMouseEvent*) – Mouse event

widgets.add_data_connection_widget

Widget shown to user when a new Data Connection is created.

author

P. Savolainen (VTT)

date 19.1.2017

Module Contents

class widgets.add_data_connection_widget.**AddDataConnectionWidget**(*toolbox*, *x*, *y*)
Bases: PySide2.QtWidgets.QWidget

A widget that queries user's preferences for a new item.

toolbox
toolbox widget

Type *ToolboxUI*

x
X coordinate of new item

Type int

y
Y coordinate of new item

Type int

connect_signals(*self*)
Connect signals to slots.

name_changed(*self*)
Update label to show upcoming folder name.

ok_clicked(*self*)
Check that given item name is valid and add it to project.

call_add_item(*self*)
Creates new Item according to user's selections.

keyPressEvent(*self*, *e*)
Close Setup form when escape key is pressed.

Parameters **e** (*QKeyEvent*) – Received key press event.

closeEvent(*self*, *event=None*)
Handle close window.

Parameters **event** (*QEvent*) – Closing event if 'X' is clicked.

widgets.add_data_interface_widget

Contains the UI and the functionality for the widget that is shown to user when a new Data Interface is created.

author

P. Savolainen (VTT)

date 13.6.2019

Module Contents

class widgets.add_data_interface_widget.**AddDataInterfaceWidget**(*toolbox*, *x*, *y*)
Bases: PySide2.QtWidgets.QWidget

A widget that queries user's preferences for a new item.

toolbox
toolbox widget

Type *ToolboxUI*

x
X coordinate of new item

Type int

y
Y coordinate of new item

Type int

connect_signals(*self*)
Connect signals to slots.

name_changed(*self*)
Update label to show upcoming folder name.

ok_clicked(*self*)
Check that given item name is valid and add it to project.

call_add_item(*self*)
Creates new Item according to user's selections.

keyPressEvent(*self*, *e*)
Close Setup form when escape key is pressed.

Parameters **e** (*QKeyEvent*) – Received key press event.

closeEvent(*self*, *event=None*)
Handle close window.

Parameters **event** (*QEvent*) – Closing event if 'X' is clicked.

widgets.add_data_store_widget

Widget shown to user when a new Data Store is created.

author

P. Savolainen (VTT)

date 19.1.2017

Module Contents

class widgets.add_data_store_widget.**AddDataStoreWidget**(*toolbox*, *x*, *y*)
Bases: PySide2.QtWidgets.QWidget

A widget that queries 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

connect_signals(*self*)
Connect signals to slots.

name_changed(*self*)
Update label to show upcoming folder name.

ok_clicked(*self*)
Check that given item name is valid and add it to project.

call_add_item(*self*)
Creates new Item according to user's selections.

keyPressEvent(*self*, *e*)
Close Setup form when escape key is pressed.

Parameters **e** (*QKeyEvent*) – Received key press event.

closeEvent(*self*, *event=None*)
Handle close window.

Parameters **event** (*QEvent*) – Closing event if 'X' is clicked.

widgets.add_tool_widget

Widget shown to user when a new Tool is created.

author

P. Savolainen (VTT)

date 19.1.2017

Module Contents

class widgets.add_tool_widget.**AddToolWidget**(*toolbox*, *x*, *y*)

Bases: PySide2.QtWidgets.QWidget

A widget that queries 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

connect_signals(*self*)

Connect signals to slots.

update_args(*self*, *row*)

Show Tool template command line arguments in text input.

Parameters **row** (*int*) – Selected row number

name_changed(*self*)

Update label to show upcoming folder name.

ok_clicked(*self*)

Check that given item name is valid and add it to project.

call_add_item(*self*)

Creates new Item according to user's selections.

keyPressEvent(*self*, *e*)

Close Setup form when escape key is pressed.

Parameters **e** (*QKeyEvent*) – Received key press event.

closeEvent(*self*, *event=None*)

Handle close window.

Parameters **event** (*QEvent*) – Closing event if 'X' is clicked.

widgets.add_view_widget

Widget shown to user when a new View is created.

author

P. Savolainen (VTT)

date 19.1.2017

Module Contents

class widgets.add_view_widget.**AddViewWidget**(*toolbox*, *x*, *y*)
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

connect_signals(*self*)
Connect signals to slots.

name_changed(*self*)
Update label to show upcoming folder name.

ok_clicked(*self*)
Check that given item name is valid and add it to project.

call_add_item(*self*)
Creates new Item according to user's selections.

keyPressEvent(*self*, *e*)
Close Setup form when escape key is pressed.

Parameters **e** (*QKeyEvent*) – Received key press event.

closeEvent(*self*, *event=None*)
Handle close window.

Parameters **event** (*QEvent*) – Closing event if 'X' is clicked.

widgets.custom_delegates

Custom item delegates.

author

M. Marin (KTH)

date 1.9.2018

Module Contents

class widgets.custom_delegates.**ComboBoxDelegate**(*parent*, *choices*)
Bases: PySide2.QtWidgets.QItemDelegate

createEditor(*self*, *parent*, *option*, *index*)

paint(*self*, *painter*, *option*, *index*)

setEditorData(*self*, *editor*, *index*)

setModelData(*self*, *editor*, *model*, *index*)

updateEditorGeometry(*self*, *editor*, *option*, *index*)

currentItemChanged(*self*)

class widgets.custom_delegates.**LineEditDelegate**
Bases: PySide2.QtWidgets.QItemDelegate

A delegate that places a fully functioning QLineEdit.

parent
either data store or spine datapackage widget

Type QMainWindow

data_committed

createEditor(*self*, *parent*, *option*, *index*)
Return CustomLineEditor. Set up a validator depending on datatype.

setEditorData(*self*, *editor*, *index*)
Init the line editor with previous data from the index.

setModelData(*self*, *editor*, *model*, *index*)
Send signal.

class widgets.custom_delegates.**CheckBoxDelegate**(*parent*, *centered=True*)
Bases: PySide2.QtWidgets.QItemDelegate

A delegate that places a fully functioning QCheckBox.

parent
either toolbox or spine datapackage widget

Type QMainWindow

centered
whether or not the checkbox should be center-aligned in the widget

Type bool

data_committed

createEditor(*self*, *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(*self*, *painter*, *option*, *index*)
Paint a checkbox without the label.

editorEvent(*self*, *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(*self*, *editor*, *model*, *index*)
Do nothing. Model data is updated by handling the *data_committed* signal.

get_checkbox_rect(*self*, *option*)

class widgets.custom_delegates.**ParameterDelegate**(*parent*)
Bases: PySide2.QtWidgets.QItemDelegate

A custom delegate for the parameter models and views in TreeViewForm.

parent
tree or graph view form

Type *DataStoreForm*

data_committed

parameter_value_editor_requested

setModelData(*self*, *editor*, *model*, *index*)
Send signal.

close_editor(*self*, *editor*, *index*, *model*)

updateEditorGeometry(*self*, *editor*, *option*, *index*)

create_parameter_value_editor(*self*, *parent*, *index*)
Returns a *CustomLineEditor* if the data from index is not of special type. Otherwise, emit the signal to request a standalone *ParameterValueEditor* from parent widget.

connect_editor_signals(*self*, *editor*, *index*)
Connect editor signals if necessary.

class widgets.custom_delegates.**ObjectParameterValueDelegate**
Bases: *widgets.custom_delegates.ParameterDelegate*

A delegate for the object parameter value model and view in TreeViewForm.

parent
tree or graph view form

Type *DataStoreForm*

createEditor(*self*, *parent*, *option*, *index*)
Return editor.

class widgets.custom_delegates.**ObjectParameterDefinitionDelegate**
Bases: *widgets.custom_delegates.ParameterDelegate*

A delegate for the object parameter definition model and view in TreeViewForm.

parent
tree or graph view form

Type *DataStoreForm*

createEditor(*self*, *parent*, *option*, *index*)
Return editor.

class widgets.custom_delegates.**RelationshipParameterValueDelegate**
Bases: *widgets.custom_delegates.ParameterDelegate*

A delegate for the relationship parameter value model and view in TreeViewForm.

parent
tree or graph view form

Type *DataStoreForm*

createEditor(*self*, *parent*, *option*, *index*)
Return editor.

class widgets.custom_delegates.**RelationshipParameterDefinitionDelegate**
Bases: *widgets.custom_delegates.ParameterDelegate*

A delegate for the object parameter definition model and view in TreeViewForm.

parent
tree or graph view form

Type *DataStoreForm*

createEditor(*self*, *parent*, *option*, *index*)
Return editor.

class widgets.custom_delegates.**ManageItemsDelegate**(*parent*)
Bases: PySide2.QtWidgets.QItemDelegate

A custom delegate for the model in {Add/Edit}ItemDialogs.

parent
parent dialog

Type *ManageItemsDialog*

data_committed

setModelData(*self*, *editor*, *model*, *index*)
Send signal.

close_editor(*self*, *editor*, *index*, *model*)

updateEditorGeometry(*self*, *editor*, *option*, *index*)

connect_editor_signals(*self*, *editor*, *index*)
Connect editor signals if necessary.

class widgets.custom_delegates.**ManageObjectClassesDelegate**(*parent*)
Bases: *widgets.custom_delegates.ManageItemsDelegate*

A delegate for the model and view in {Add/Edit}ObjectClassesDialog.

parent
parent dialog

Type *ManageItemsDialog*

icon_color_editor_requested

createEditor(*self*, *parent*, *option*, *index*)
Return editor.

paint(*self*, *painter*, *option*, *index*)
Get a pixmap from the index data and paint it in the middle of the cell.

class widgets.custom_delegates.**ManageObjectsDelegate**
Bases: *widgets.custom_delegates.ManageItemsDelegate*

A delegate for the model and view in {Add/Edit}ObjectsDialog.

parent
parent dialog

Type *ManageItemsDialog*

createEditor(*self*, *parent*, *option*, *index*)
Return editor.

class widgets.custom_delegates.**ManageRelationshipClassesDelegate**
Bases: *widgets.custom_delegates.ManageItemsDelegate*

A delegate for the model and view in {Add/Edit}RelationshipClassesDialog.

parent
parent dialog

Type *ManageItemsDialog*

createEditor(*self*, *parent*, *option*, *index*)
Return editor.

class widgets.custom_delegates.**ManageRelationshipsDelegate**
Bases: *widgets.custom_delegates.ManageItemsDelegate*

A delegate for the model and view in {Add/Edit}RelationshipsDialog.

parent
parent dialog

Type *ManageItemsDialog*

createEditor(*self*, *parent*, *option*, *index*)
Return editor.

class widgets.custom_delegates.**RemoveTreeItemsDelegate**
Bases: *widgets.custom_delegates.ManageItemsDelegate*

A delegate for the model and view in RemoveTreeItemsDialog.

parent
parent dialog

Type *ManageItemsDialog*

createEditor(*self*, *parent*, *option*, *index*)
Return editor.

class widgets.custom_delegates.**ManageParameterTagsDelegate**
Bases: *widgets.custom_delegates.ManageItemsDelegate*

A delegate for the model and view in ManageParameterTagsDialog.

parent
parent dialog

Type *ManageItemsDialog*

createEditor(*self*, *parent*, *option*, *index*)
Return editor.

class widgets.custom_delegates.**ForeignKeysDelegate**(*parent*)
Bases: PySide2.QtWidgets.QItemDelegate

A QComboBox delegate with checkboxes.

parent
spine datapackage widget

Type *SpineDatapackageWidget*

data_committed

close_field_name_list_editor(*self*, *editor*, *index*, *model*)

createEditor(*self*, *parent*, *option*, *index*)
Return editor.

setEditorData(*self*, *editor*, *index*)
Set editor data.

setModelData(*self*, *editor*, *model*, *index*)
Send signal.

widgets.custom_editors

Custom editors for model/view programming.

author

M. Marin (KTH)

date 2.9.2018

Module Contents

class widgets.custom_editors.**CustomLineEditor**
Bases: PySide2.QtWidgets.QLineEdit

A custom QLineEdit to handle data from models.

parent
the widget that wants to edit the data

Type QWidget

set_data(*self*, *data*)

data(*self*)

keyPressEvent(*self*, *event*)
Don't allow shift key to clear the contents.

class widgets.custom_editors.**CustomComboEditor**
Bases: PySide2.QtWidgets.QComboBox

A custom QComboBox to handle data from models.

parent
the widget that wants to edit the data

Type QWidget

data_committed

set_data(*self*, *current_text*, *items*)

data(*self*)

class widgets.custom_editors.**CustomLineEditDelegate**(*parent*)
Bases: PySide2.QtWidgets.QItemDelegate

A delegate for placing a CustomLineEditor on the first row of SearchBarEditor.

parent
search bar editor

Type *SearchBarEditor*

text_edited

setModelData(*self*, *editor*, *model*, *index*)

createEditor(*self*, *parent*, *option*, *index*)
Create editor and 'forward' *textEdited* signal.

eventFilter(*self*, *editor*, *event*)
Handle all sort of special cases.

class widgets.custom_editors.**SearchBarEditor**(*parent*, *elder_sibling=None*, *is_json=False*)
Bases: PySide2.QtWidgets.QTableView

A Google-like search bar, implemented as a QTableView with a CustomLineEditDelegate in the first row.

parent
the parent for this widget

Type QWidget

elder_sibling
another widget which is used to find this widget's position.

Type QWidget or NoneType

data_committed

set_data(*self*, *current*, *all_data*)
Populate model and initialize first index.

set_base_size(*self*, *size*)

update_geometry(*self*)
Update geometry. Resize the widget to optimal size, then adjust its position.

refit(*self*)
Resize to optimal size.

data(*self*)

_handle_delegate_text_edited(*self*, *text*)
Filter model as the first row is being edited.

_proxy_model_filter_accepts_row(*self*, *source_row*, *source_parent*)
Overridden method to always accept first row.

keyPressEvent(*self*, *event*)
Set data from current index into first index as the user navigates through the table using the up and down keys.

currentChanged(*self*, *current*, *previous*)

edit_first_index(*self*)
Edit first index if valid and not already being edited.

mouseMoveEvent(*self*, *event*)
Make hovered index the current index.

mousePressEvent(*self*, *event*)
Commit data.

class widgets.custom_editors.**SearchBarDelegate**(*parent*)
Bases: PySide2.QtWidgets.QItemDelegate

A custom delegate to place a SearchBarEditor on each cell of a MultiSearchBarEditor.

parent
multi search bar editor

Type *MultiSearchBarEditor*

data_committed

setModelData(*self*, *editor*, *model*, *index*)

createEditor(*self*, *parent*, *option*, *index*)

updateEditorGeometry(*self*, *editor*, *option*, *index*)

close_editor(*self*, *editor*, *index*, *model*)

eventFilter(*self*, *editor*, *event*)

class widgets.custom_editors.**MultiSearchBarEditor**(*parent*, *elder_sibling=None*)
Bases: PySide2.QtWidgets.QTableView

A table view made of several Google-like search bars.

set_data(*self*, *header*, *currents*, *alls*)

data(*self*)

set_base_size(*self*, *size*)

update_geometry(*self*)
Update geometry.

start_editing(*self*)
Start editing first item.

class widgets.custom_editors.**CheckListEditor**(*parent*, *elder_sibling=None*)
Bases: PySide2.QtWidgets.QTableView

A check list editor.

keyPressEvent(*self*, *event*)
Toggle checked state.

toggle_checked_state(*self*, *index*)

mouseMoveEvent(*self*, *event*)
Highlight current row.

mousePressEvent(*self*, *event*)
Toggle checked state.

set_data(*self*, *item_names*, *current_item_names*)
Set data and update geometry.

data(*self*)

set_base_size(*self*, *size*)

update_geometry(*self*)
Update geometry.

class widgets.custom_editors.**JSONEditor**(*parent*, *elder_sibling*, *popup=False*)
Bases: PySide2.QtWidgets.QTabWidget

A double JSON editor, featuring: - A QTextEdit for editing arbitrary json. - A QTableView for editing json array.

data_committed

_view_key_press_event(*self*, *event*)
Accept key events on the view to avoid weird behaviour, when trying to navigate outside of its limits.

eventFilter(*self*, *widget*, *event*)
Intercept events to text_edit and table_view to enable consistent behavior.

check_focus(*self*)
Called when either the text edit or the table view lose focus. Check if the focus is still on this widget (which would mean it was a tab change) otherwise emit signal so this is closed.

_handle_current_changed(*self*, *index*)
Update json data on text edit or table view, and set focus.

set_data(*self*, *data*, *current_index*)
Set data on text edit or table view (model) depending on current index.

start_editing(*self*)
Start editing.

set_base_size(*self*, *size*)

update_geometry(*self*)
Update geometry.

data(*self*)

class widgets.custom_editors.**IconPainterDelegate**
Bases: PySide2.QtWidgets.QItemDelegate

A delegate to highlight decorations in a QListWidget.

paint(*self*, *painter*, *option*, *index*)
Highlight selected items.

class 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.

_proxy_model_filter_accepts_row(*self*, *source_row*, *source_parent*)
Overridden method to filter icons according to search terms.

connect_signals(*self*)
Connect signals to slots.

set_data(*self*, *data*)

data(*self*)

class widgets.custom_editors.**NumberParameterInlineEditor**(*parent*)
Bases: PySide2.QtWidgets.QDoubleSpinBox

An editor widget for numeric (datatype double) parameter values.

set_data(*self*, *data*)

data(*self*)

widgets.custom_menus

Classes for custom context menus and pop-up menus.

author

P. Savolainen (VTT)

date 9.1.2018

Module Contents

widgets.custom_menus.**handle_plotting_failure**(*error*)
Reports a PlottingError exception to the user.

class widgets.custom_menus.**CustomContextMenu**(*parent*)
Bases: PySide2.QtWidgets.QMenu

Context menu master class for several context menus.

parent
Parent for menu widget (ToolboxUI)

Type QWidget

add_action(*self*, *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(*self*, *option*)
Sets the action which was clicked.

Parameters **option** (*str*) – string with the text description of the action

get_action(*self*)
Returns the clicked action, a string with a description.

class widgets.custom_menus.**ProjectItemContextMenu**(*parent*, *position*, *index*)
Bases: *widgets.custom_menus.CustomContextMenu*

Context menu for project items both in the QTreeView and in the QGraphicsView.

parent
Parent for menu widget (ToolboxUI)

Type QWidget

position
Position on screen

Type QPoint

index
Index of item that requested the context-menu

Type QModelIndex

class widgets.custom_menus.**LinkContextMenu**(*parent*, *position*, *index*, *parallel_link=None*)
Bases: *widgets.custom_menus.CustomContextMenu*

Context menu class for connection links.

parent
Parent for menu widget (ToolboxUI)

Type QWidget

position
Position on screen

Type QPoint

index
Index of item that requested the context-menu

Type QModelIndex

parallel_link
Link that is parallel to the one that requested the menu

Type *Link*(QGraphicsPathItem)

class widgets.custom_menus.**ToolTemplateContextMenu**(*parent*, *position*, *index*)
Bases: *widgets.custom_menus.CustomContextMenu*

Context menu class for Tool templates.

parent
Parent for menu widget (ToolboxUI)

Type QWidget

position
Position on screen

Type QPoint

index
Index of item that requested the context-menu

Type QModelIndex

class widgets.custom_menus.**DcRefContextMenu**(*parent*, *position*, *index*)
Bases: *widgets.custom_menus.CustomContextMenu*

Context menu class for references view in Data Connection properties.

parent
Parent for menu widget (ToolboxUI)

Type QWidget

position
Position on screen

Type QPoint

index
Index of item that requested the context-menu

Type QModelIndex

class widgets.custom_menus.**DcDataContextMenu**(*parent*, *position*, *index*)
Bases: *widgets.custom_menus.CustomContextMenu*

Context menu class for data view in Data Connection properties.

parent
Parent for menu widget (ToolboxUI)

Type QWidget

position
Position on screen

Type QPoint

index
Index of item that requested the context-menu

Type QModelIndex

class widgets.custom_menus.**ToolPropertiesContextMenu**(*parent*, *position*, *index*)
Bases: *widgets.custom_menus.CustomContextMenu*

Common context menu class for all Tool QTreeViews in Tool properties.

parent
Parent for menu widget (ToolboxUI)

Type QWidget

position
Position on screen

Type QPoint

index
Index of item that requested the context-menu

Type QModelIndex

class widgets.custom_menus.**ViewPropertiesContextMenu**(*parent*, *position*, *index*)
Bases: *widgets.custom_menus.CustomContextMenu*

Context menu class for the references tree view of the View project item properties.

parent
Parent for menu widget (ToolboxUI)

Type QWidget

position
Position on screen

Type QPoint

index
Index of item that requested the context-menu

Type QModelIndex

class widgets.custom_menus.**DiFilesContextMenu**(*parent*, *position*, *index*)
Bases: *widgets.custom_menus.CustomContextMenu*

Context menu class for source files view in Data Interface properties.

parent
Parent for menu widget (ToolboxUI)

Type QWidget

position
Position on screen

Type QPoint

index
Index of item that requested the context-menu

Type QModelIndex

class widgets.custom_menus.**ObjectTreeContextMenu**(*parent*, *position*, *index*)
Bases: *widgets.custom_menus.CustomContextMenu*

Context menu class for object tree items in tree view form.

parent
Parent for menu widget (TreeViewForm)

Type QWidget

position
Position on screen

Type QPoint

index
Index of item that requested the context-menu

Type QModelIndex

class widgets.custom_menus.**RelationshipTreeContextMenu**(*parent*, *position*, *index*)
Bases: *widgets.custom_menus.CustomContextMenu*

Context menu class for relationship tree items in tree view form.

parent
Parent for menu widget (TreeViewForm)

Type QWidget

position
Position on screen

Type QPoint

index
Index of item that requested the context-menu

Type QModelIndex

class widgets.custom_menus.**ParameterContextMenu**(*parent*, *position*, *index*)
Bases: *widgets.custom_menus.CustomContextMenu*

Context menu class for object (relationship) parameter items in tree views.

parent
Parent for menu widget (TreeViewForm)

Type QWidget

position
Position on screen

Type QPoint

index
Index of item that requested the context-menu

Type QModelIndex

class widgets.custom_menus.**SimpleEditableParameterValueContextMenu**(*parent*, *position*, *index*)
Bases: *widgets.custom_menus.CustomContextMenu*

Context menu class for object (relationship) parameter value items in graph views.

parent
Parent for menu widget (TreeViewForm)

Type QWidget

position
Position on screen

Type QPoint

index
Index of item that requested the context-menu

Type QModelIndex

class widgets.custom_menus.**EditableParameterValueContextMenu**(*parent*, *position*, *index*)
Bases: *widgets.custom_menus.CustomContextMenu*

Context menu class for object (relationship) parameter value items in tree views.

parent
Parent for menu widget (TreeViewForm)

Type QWidget

position
Position on screen

Type QPoint

index
Index of item that requested the context-menu

Type QModelIndex

class widgets.custom_menus.**ParameterValueListContextMenu**(*parent*, *position*, *index*)
Bases: *widgets.custom_menus.CustomContextMenu*

Context menu class for parameter enum view in tree view form.

parent
Parent for menu widget (TreeViewForm)

Type QWidget

position
Position on screen

Type QPoint

index
Index of item that requested the context-menu

Type QModelIndex

class widgets.custom_menus.**GraphViewContextMenu**(*parent*, *position*)
Bases: *widgets.custom_menus.CustomContextMenu*

Context menu class for qgraphics view in graph view.

parent
Parent for menu widget (GraphViewForm)

Type QWidget

position
Position on screen

Type QPoint

class widgets.custom_menus.**ObjectItemContextMenu**(*parent*, *position*, *graphics_item*)
Bases: *widgets.custom_menus.CustomContextMenu*

Context menu class for object graphic items in graph view.

parent
Parent for menu widget (GraphViewForm)

Type QWidget

position
Position on screen

Type QPoint

graphics_item
item that requested the menu

Type *ObjectItem* (QGraphicsItem)

class widgets.custom_menus.**CustomPopupMenu**(*parent*)
Bases: PySide2.QtWidgets.QMenu

Popup menu master class for several popup menus.

parent
Parent widget of this pop-up menu

Type QWidget

add_action(*self*, *text*, *slot*, *enabled=True*)
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?

class widgets.custom_menus.**AddToolTemplatePopupMenu**(*parent*)
Bases: *widgets.custom_menus.CustomPopupMenu*

Popup menu class for add Tool template button.

parent
parent widget (ToolboxUI)

Type QWidget

class widgets.custom_menus.**ToolTemplateOptionsPopupMenu**(*parent*, *tool*)
Bases: *widgets.custom_menus.CustomPopupMenu*

Popup menu class for tool template options button in Tool item.

parent
Parent widget of this menu (ToolboxUI)

Type QWidget

tool
Tool item that is associated with the pressed button

Type *Tool*

class widgets.custom_menus.**AddIncludesPopupMenu**(*parent*)
Bases: *widgets.custom_menus.CustomPopupMenu*

Popup menu class for add includes button in Tool Template widget.

parent
Parent widget (ToolTemplateWidget)

Type QWidget

class widgets.custom_menus.**CreateMainProgramPopupMenu**(*parent*)
Bases: *widgets.custom_menus.CustomPopupMenu*

Popup menu class for add main program QToolButton in Tool Template editor.

parent
Parent widget (ToolTemplateWidget)

Type QWidget

class widgets.custom_menus.**FilterMenu**(*parent=None*, *show_empty=True*)
Bases: PySide2.QtWidgets.QMenu

Filter menu to use together with FilterWidget in TabularViewForm.

filterChanged

add_items_to_filter_list(*self*, *items*)

remove_items_from_filter_list(*self*, *items*)

set_filter_list(*self*, *data*)

_clear_filter(*self*)

_check_filter(*self*)

_cancel_filter(*self*)

_change_filter(*self*)

class widgets.custom_menus.**PivotTableModelMenu**(*model*, *proxy_model*, *parent=None*)
Bases: PySide2.QtWidgets.QMenu

_find_selected_indexes(*self*, *indexes*)
Find any selected index values

_find_selected_relationships(*self*, *indexes*)
Find any selected tuple combinations in self.relationship_tuple_key

_get_selected_indexes(*self*)
Find selected indexes of parent, map to source if proxy is given

delete_invalid_row(*self*)

delete_invalid_col(*self*)

insert_row(*self*)

insert_col(*self*)

delete_values(*self*)
deletes selected indexes in pivot_table

restore_values(*self*)
restores edited selected indexes in pivot_table

delete_index_values(*self*)
finds selected index items and deletes

delete_relationship_values(*self*)
finds selected relationships deletes

open_value_editor(*self*)
Opens the parameter value editor for the first selected cell.

plot(*self*)
Plots the selected cells in the pivot table.

request_menu(*self*, *QPos=None*)
Shows the context menu on the screen.

class widgets.custom_menus.**PivotTableHorizontalHeaderMenu**(*model*, *parent=None*)
Bases: PySide2.QtWidgets.QMenu

A context menu for the horizontal header of a pivot table.

model
a model

Type *PivotTableModel*

parent
a parent widget

Type QWidget

_plot_column(*self*)
Plots a single column not the selection.

request_menu(*self*, *pos*)
Shows the context menu on the screen.

_set_x_flag(*self*)
Sets the X flag for a column.

widgets.custom_qdialog

Classes for custom QDialogs to add and edit database items.

author

M. Marin (KTH)

date 13.5.2018

Module Contents

class widgets.custom_qdialog.**ManageItemsDialog**(*parent*)
Bases: PySide2.QtWidgets.QDialog

A dialog with a CopyPasteTableView and a QDialogButtonBox, to be extended into dialogs to query user's preferences for adding/editing/managing data items

parent
data store widget

Type *TreeViewForm*

connect_signals(*self*)
Connect signals to slots.

resize_window_to_columns(*self*)

_handle_model_data_changed(*self*, *top_left*, *bottom_right*, *roles*)
Reimplement in subclasses to handle changes in model data.

set_model_data(*self*, *index*, *data*)
Update model data.

_handle_model_reset(*self*)
Resize columns and form.

class widgets.custom_qdialog.**AddItemsDialog**(*parent*)
Bases: *widgets.custom_qdialog.ManageItemsDialog*

connect_signals(*self*)

remove_selected_rows(*self*, *checked=True*)

all_databases(*self*, *row*)
Returns a list of db names available for a given row. Used by delegates.

class widgets.custom_qdialog.**GetObjectClassesMixin**
Provides a method to retrieve object classes for AddObjectsDialog and AddRelationshipClassesDialog.

object_class_name_list(*self*, *row*)
Return a list of object class names present in all databases selected for given row. Used by *ManageObjectsDelegate*.

class widgets.custom_qdialog.**GetObjectsMixin**
Provides a method to retrieve objects for AddRelationshipsDialog and EditRelationshipsDialog.

object_name_list(*self*, *row*, *column*)
Return a list of object names present in all databases selected for given row. Used by *ManageRelationshipsDelegate*.

class widgets.custom_qdialog.**ShowIconColorEditorMixin**
Provides methods to show an *IconColorEditor* upon request.

create_object_pixmap(*self*, *display_icon*)

show_icon_color_editor(*self*, *index*)

class widgets.custom_qdialog.**AddObjectClassesDialog**(*parent*)
Bases: *widgets.custom_qdialog.AddItemsDialog*, *widgets.custom_qdialog.ShowIconColorEditorMixin*

A dialog to query user's preferences for new object classes.

parent
data store widget

Type *DataStoreForm*

connect_signals(*self*)

accept(*self*)
Collect info from dialog and try to add items.

class widgets.custom_qdialog.**AddObjectsDialog**(*parent*, *class_name=None*, *force_default=False*)

Bases: *widgets.custom_qdialog.AddItemsDialog*, *widgets.custom_qdialog.GetObjectClassesMixin*

A dialog to query user's preferences for new objects.

parent
data store widget

Type *DataStoreForm*

class_name
default object class name

Type str

force_default
if True, defaults are non-editable

Type bool

_handle_model_data_changed(*self*, *top_left*, *bottom_right*, *roles*)
Set decoration role data.

accept(*self*)
Collect info from dialog and try to add items.

class widgets.custom_qdialog.**AddRelationshipClassesDialog**(*parent*, *object_class_one_name=None*, *force_default=False*)

Bases: *widgets.custom_qdialog.AddItemsDialog*, *widgets.custom_qdialog.GetObjectClassesMixin*

A dialog to query user's preferences for new relationship classes.

parent
data store widget

Type *DataStoreForm*

object_class_one_name
default object class name to put in first dimension

Type str

force_default
if True, defaults are non-editable

Type bool

connect_signals(*self*)
Connect signals to slots.

_handle_spin_box_value_changed(*self*, *i*)

insert_column(*self*)

remove_column(*self*)

_handle_model_data_changed(*self*, *top_left*, *bottom_right*, *roles*)

accept(*self*)
Collect info from dialog and try to add items.

class widgets.custom_qdialog.**AddRelationshipsDialog**(*parent*, *relationship_class_key=None*, *object_class_name=None*, *object_name=None*, *force_default=False*)

Bases: *widgets.custom_qdialog.AddItemsDialog*, *widgets.custom_qdialog.GetObjectsMixin*

A dialog to query user's preferences for new relationships.

parent
data store widget

Type *DataStoreForm*

relationship_class_key
(class_name, object_class_name_list) for identifying the relationship class

Type tuple

object_name
default object name

Type str

object_class_name
default object class name

Type str

force_default
if True, defaults are non-editable

Type bool

connect_signals(*self*)
Connect signals to slots.

call_reset_model(*self*, *text*)
Called when relationship class's combobox's index changes. Update relationship_class attribute accordingly and reset model.

reset_model(*self*)
Setup model according to current relationship class selected in combobox.

_handle_model_data_changed(*self*, *top_left*, *bottom_right*, *roles*)

accept(*self*)
Collect info from dialog and try to add items.

class widgets.custom_qdialog.**EditOrRemoveItemsDialog**(*parent*)

Bases: *widgets.custom_qdialog.ManageItemsDialog*

all_databases(*self*, *row*)
Returns a list of db names available for a given row. Used by delegates.

class widgets.custom_qdialog.**EditObjectClassesDialog**(*parent*, *db_map_dicts*)

Bases: *widgets.custom_qdialog.EditOrRemoveItemsDialog*, *widgets.custom_qdialog.ShowIconColorEditorMixin*

A dialog to query user's preferences for updating object classes.

parent
data store widget

Type *DataStoreForm*

db_map_dicts
list of dictionaries mapping dbs to object classes for editing

Type list

connect_signals(*self*)

accept(*self*)
Collect info from dialog and try to update items.

class widgets.custom_qdialog.**EditObjectsDialog**(*parent*, *db_map_dicts*)
Bases: *widgets.custom_qdialog.EditOrRemoveItemsDialog*

A dialog to query user's preferences for updating objects.

parent
data store widget

Type *DataStoreForm*

db_map_dicts
list of dictionaries mapping dbs to objects for editing

Type list

accept(*self*)
Collect info from dialog and try to update items.

class widgets.custom_qdialog.**EditRelationshipClassesDialog**(*parent*, *db_map_dicts*)
Bases: *widgets.custom_qdialog.EditOrRemoveItemsDialog*

A dialog to query user's preferences for updating relationship classes.

parent
data store widget

Type *DataStoreForm*

db_map_dicts
list of dictionaries mapping dbs to relationship classes for editing

Type list

accept(*self*)
Collect info from dialog and try to update items.

class widgets.custom_qdialog.**EditRelationshipsDialog**(*parent*, *db_map_dicts*, *ref_class_key*)
Bases: *widgets.custom_qdialog.EditOrRemoveItemsDialog*, *widgets.custom_qdialog.GetObjectsMixin*

A dialog to query user's preferences for updating relationships.

parent
data store widget

Type *DataStoreForm*

db_map_dicts
list of dictionaries mapping dbs to relationships for editing

Type list

ref_class_key
(class_name, object_class_name_list) for identifying the relationship class

Type tuple

accept(*self*)
Collect info from dialog and try to update items.

class widgets.custom_qdialog.**RemoveTreeItemsDialog**(*parent*, ***kwargs*)
Bases: *widgets.custom_qdialog.EditOrRemoveItemsDialog*

A dialog to query user's preferences for removing tree items.

parent
data store widget

Type *TreeViewForm*

accept(*self*)
Collect info from dialog and try to remove items.

class widgets.custom_qdialog.**ManageParameterTagsDialog**(*parent*)
Bases: *widgets.custom_qdialog.ManageItemsDialog*

A dialog to query user's preferences for managing parameter tags.

parent
data store widget

Type *TreeViewForm*

create_check_boxes(*self*, *start*, *stop*)
Create check boxes in remove column.

all_databases(*self*, *row*)
Returns a list of db names available for a given row. Used by delegates.

accept(*self*)
Collect info from dialog and try to update, remove, add items.

class widgets.custom_qdialog.**CommitDialog**(*parent*, **db_names*)
Bases: PySide2.QtWidgets.QDialog

A dialog to query user's preferences for new commit.

parent
data store widget

Type *TreeViewForm*

db_names
database names

Type Iterable

receive_text_changed(*self*)
Called when text changes in the commit msg text edit. Enable/disable commit button accordingly.

widgets.custom_qgraphicsscene

Custom QGraphicsScene used in the Design View.

author

P. Savolainen (VTT)

date 13.2.2019

Module Contents

class widgets.custom_qgraphicsscene.**CustomQGraphicsScene**(*parent*, *toolbox*)
Bases: PySide2.QtWidgets.QGraphicsScene

A scene that handles drag and drop events of DraggableWidget sources.

files_dropped_on_dc

connect_signals(*self*)
Connect scene signals.

resize_scene(*self*)
Resize scene to be at least the size of items bounding rectangle. Does not let the scene shrink.

scene_changed(*self*, *rects*)
Resize scene as it changes.

handle_selection_changed(*self*)
Synchronize selection with the project tree.

set_bg_color(*self*, *color*)
Change background color when this is changed in Settings.

Parameters **color** (*QColor*) – Background color

set_bg_grid(*self*, *bg*)
Enable or disable background grid.

Parameters **bg** (*boolean*) – True to draw grid, False to fill background with a solid color

dragLeaveEvent(*self*, *event*)
Accept event.

dragEnterEvent(*self*, *event*)
Accept event. Then call the super class method only if drag source is not a DraggableWidget (from Add Item toolbar).

dragMoveEvent(*self*, *event*)
Accept event. Then call the super class method only if drag source is not a DraggableWidget (from Add Item toolbar).

dropEvent(*self*, *event*)
Only accept drops when the source is an instance of DraggableWidget (from Add Item toolbar). Capture text from event's mimedata and show the appropriate 'Add Item form.'

drawBackground(*self*, *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

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

class widgets.custom_qgraphicsviews.**CustomQGraphicsView**(*parent*)
Bases: PySide2.QtWidgets.QGraphicsView

Super class for Design and Graph QGraphicsViews.

parent
Parent widget

Type QWidget

keyPressEvent(*self*, *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

enterEvent(*self*, *event*)
Overridden method. Do not show the stupid open hand mouse cursor.

Parameters **event** (*QEvent*) – event

mousePressEvent(*self*, *event*)
Set rubber band selection mode if Control pressed. Enable resetting the zoom factor from the middle mouse button.

mouseReleaseEvent(*self*, *event*)
Reestablish scroll hand drag mode.

wheelEvent(*self*, *event*)
Zoom in/out.

Parameters **event** (*QWheelEvent*) – Mouse wheel event

resizeEvent(*self*, *event*)
Updates zoom if needed when the view is resized.

Parameters **event** (*QResizeEvent*) – a resize event

setScene(*self*, *scene*)
Sets a new scene to this view.

Parameters **scene** (*QGraphicsScene*) – a new scene

_update_zoom_limits(*self*, *rect*)
Updates the minimum zoom limit and the zoom level with which the entire scene fits the view.

Parameters **rect** (*QRectF*) – the scene's rect

scaling_time(*self*, *pos*)
Called when animation value for smooth zoom changes. Perform zoom.

anim_finished(*self*)
Called when animation for smooth zoom finishes. Clean up.

zoom_in(*self*)
Perform a zoom in with a fixed scaling.

zoom_out(*self*)
Perform a zoom out with a fixed scaling.

reset_zoom(*self*)
Reset zoom to the default factor.

gentle_zoom(*self*, *factor*, *zoom_focus*)
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

class widgets.custom_qgraphicsviews.**DesignQGraphicsView**(*parent*)
Bases: *widgets.custom_qgraphicsviews.CustomQGraphicsView*

QGraphicsView for the Design View.

parent
Graph View Form's (QMainWindow) central widget (self.centralwidget)

Type QWidget

mousePressEvent(*self*, *event*)
Manage drawing of links. Handle the case where a link is being drawn and the user doesn't hit a connector button.

Parameters **event** (*QGraphicsSceneMouseEvent*) – Mouse event

mouseMoveEvent(*self*, *event*)
Update line end position.

Parameters **event** (*QGraphicsSceneMouseEvent*) – Mouse event

set_ui(*self*, *toolbox*)
Set a new scene into the Design View when app is started.

init_scene(*self*, *empty=False*)
Resize scene and add a link drawer on scene. The scene must be cleared before calling this.

Parameters **empty** (*boolean*) – True when creating a new project

set_project_item_model(*self*, *model*)
Set project item model.

set_connection_model(*self*, *model*)
Set connection model and connect signals.

add_link(*self*, *src_connector*, *dst_connector*, *index*)
Draws link between source and sink items on scene and appends connection model. Refreshes View references if needed.

Parameters

- **src_connector** (ConnectorButton) – Source connector button
- **dst_connector** (ConnectorButton) – Destination connector button
- **index** (*QModelIndex*) – Index in connection model

remove_link(*self*, *index*)
Removes link between source and sink items on scene and updates connection model. Refreshes View references if needed.

take_link(*self*, *index*)
Remove link, then start drawing another one from the same source connector.

restore_links(*self*)
Iterates connection model and draws links for each valid entry. Should be called only when a project is loaded from a save file.

connection_rows_removed(*self*, *index*, *first*, *last*)
Update view when connection model changes.

connection_columns_removed(*self*, *index*, *first*, *last*)
Update view when connection model changes.

draw_links(*self*, *connector*)
Draw links when slot button is clicked.

Parameters **connector** (ConnectorButton) – Connector button that triggered the drawing

emit_connection_information_message(*self*)
Inform user about what connections are implemented and how they work.

class widgets.custom_qgraphicsviews.**GraphQGraphicsView**(*parent*)
Bases: *widgets.custom_qgraphicsviews.CustomQGraphicsView*

QGraphicsView for the Graph View.

item_dropped

dragLeaveEvent(*self*, *event*)
Accept event. Then call the super class method only if drag source is not DragListView.

dragEnterEvent(*self*, *event*)
Accept event. Then call the super class method only if drag source is not DragListView.

dragMoveEvent(*self*, *event*)
Accept event. Then call the super class method only if drag source is not DragListView.

dropEvent(*self*, *event*)
Only accept drops when the source is an instance of DragListView. Capture text from event's mimedata and emit signal.

contextMenuEvent(*self*, *e*)
Show context menu.

Parameters **e** (*QContextMenuEvent*) – Context menu event

widgets.custom_qlineedit

Classes for custom line edits.

author

M. Marin (KTH)

date 11.10.2018

Module Contents

class widgets.custom_qlineedit.**CustomQLineEdit**
Bases: PySide2.QtWidgets.QLineEdit

A custom QLineEdit that accepts file drops and displays the path.

parent
Parent for line edit widget (DataStoreWidget)

Type QMainWindow

file_dropped

dragEnterEvent (*self*, *event*)
Accept a single file drop from the filesystem.

dragMoveEvent (*self*, *event*)
Accept event.

dropEvent (*self*, *event*)
Emit file_dropped signal with the file for the dropped url.

widgets.custom_qlistview

Classes for custom QListView.

author

M. Marin (KTH)

date 14.11.2018

Module Contents

class widgets.custom_qlistview.**DragListView**(*parent*)
Bases: PySide2.QtWidgets.QListView

Custom QListView class with dragging support.

parent
The parent of this view

Type QWidget

mousePressEvent (*self*, *event*)
Register drag start position

mouseMoveEvent (*self*, *event*)
Start dragging action if needed

mouseReleaseEvent (*self*, *event*)
Forget drag start position

class widgets.custom_qlistview.**TestListView**(*parent=None*)
Bases: PySide2.QtWidgets.QListWidget

afterDrop

allowedDragLists = []

dragEnterEvent (*self*, *event*)

dropEvent (*self*, *event*)

widgets.custom_qtableview

Custom QTableView classes that support copy-paste and the like.

author

M. Marin (KTH)

date 18.5.2018

Module Contents

class widgets.custom_qtableview.**CopyPasteTableView**
Bases: PySide2.QtWidgets.QTableView

Custom QTableView class with copy and paste methods.

keyPressEvent(*self*, *event*)
Copy and paste to and from clipboard in Excel-like format.

copy(*self*)
Copy current selection to clipboard in excel format.

canPaste(*self*)

paste(*self*)
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

paste_on_selection(*self*)
Paste clipboard data on selection, but not beyond. If data is smaller than selection, repeat data to fit selection.

paste_normal(*self*)
Paste clipboard data, overwriting cells if needed

class widgets.custom_qtableview.**AutoFilterMenu**(*parent*)
Bases: PySide2.QtWidgets.QMenu

A widget to show the auto filter 'menu'.

parent
the parent widget.

Type QTableView

asc_sort_triggered

desc_sort_triggered

filter_triggered

_model_flags(*self*, *index*)
Return no item flags.

_model_data(*self*, *index*, *role=Qt.DisplayRole*)
Read checked state from first column.

_proxy_model_filter_accepts_row(*self*, *source_row*, *source_parent*)
Overridden method to always accept first row.

_handle_view_entered(*self*, *index*)
Highlight current row.

_view_key_press_event(*self*, *event*)

_handle_view_clicked(*self*, *index*)

toggle_checked_state(*self*, *checked_index*)
Toggle checked state.

_view_leave_event(*self*, *event*)
Clear selection.

set_data_for_all_index(*self*)
Set data for 'all' index based on data from all other indexes.

_handle_ok_action_triggered(*self*, *checked=False*)
Called when user presses Ok.

set_values(*self*, *values*)
Set values to show in the 'menu'.

popup(*self*, *pos*, *width=0*, *at_action=None*)

class widgets.custom_qtableview.**AutoFilterCopyPasteTableView**(*parent*)
Bases: *widgets.custom_qtableview.CopyPasteTableView*

Custom QTableView class with autofilter functionality.

parent
The parent of this view

Type QWidget

filter_changed

keyPressEvent(*self*, *event*)

setModel(*self*, *model*)
Disconnect sectionPressed signal, only connect it to show_filter_menu slot. Otherwise the column is selected when pressing on the header.

toggle_auto_filter(*self*, *logical_index*)
Called when user clicks on a horizontal section header. Show/hide the auto filter widget.

update_auto_filter(*self*)
Called when the user selects Ok in the auto filter menu. Set 'filtered out values' in auto filter model.

sort_model_ascending(*self*)
Called when the user selects sort ascending in the auto filter widget.

sort_model_descending(*self*)
Called when the user selects sort descending in the auto filter widget.

class widgets.custom_qtableview.**FrozenTableView**(*parent=None*)
Bases: PySide2.QtWidgets.QTableView

clear(*self*)

get_selected_row(*self*)

set_data(*self*, *headers*, *values*)

class widgets.custom_qtableview.**SimpleCopyPasteTableView**(*parent=None*)
Bases: PySide2.QtWidgets.QTableView

Custom QTableView class that copies and paste data in response to key press events.

parent
The parent of this view

Type QWidget

clipboard_data_changed(*self*)

keyPressEvent(*self*, *event*)
Copy and paste to and from clipboard in Excel-like format.

class widgets.custom_qtableview.**IndexedParameterValueTableViewBase**
Bases: *widgets.custom_qtableview.CopyPasteTableView*

Custom QTableView base class with copy and paste methods for indexed parameter values.

copy(*self*)
Copy current selection to clipboard in CSV format.

static _read_pasted_text(*text*)
Reads CSV formatted table.

paste(*self*)
Pastes data from clipboard to selection.

static _range(*indexes*)
Returns the top left and bottom right corners of selected model indexes.

Parameters **indexes** (*list*) – a list of selected QModelIndex objects

Returns a tuple (top row, bottom row, left column, right column)

_select_pasted(*self*, *indexes*)
Selects the given model indexes.

class widgets.custom_qtableview.**TimeSeriesFixedResolutionTableView**
Bases: *widgets.custom_qtableview.IndexedParameterValueTableViewBase*

A QTableView for fixed resolution time series table.

paste(*self*)
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

_paste_to_values_column(*self*, *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))

class widgets.custom_qtableview.**IndexedValueTableView**
Bases: *widgets.custom_qtableview.IndexedParameterValueTableViewBase*

A QTableView class with for variable resolution time series and time patterns.

paste(*self*)
Pastes data from clipboard.

_paste_two_columns(*self*, *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)

_paste_single_column(*self*, *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)

static _read_pasted_text(*text*)
Parses a given CSV table

Parameters **text** (*str*) – a CSV table

Returns a tuple (data indexes, data values)

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

class widgets.custom_qtextbrowser.**CustomQTextBrowser**(*parent*)
Bases: PySide2.QtWidgets.QTextBrowser

Custom QTextBrowser class.

parent
Parent widget

Type QWidget

max_blocks
Returns the upper limit of text blocks that can be appended to the widget.

append(*self*, *text*)
Appends new text block to the end of the current contents.

If the widget contains more text blocks after the addition than a set limit, blocks will be deleted at the start of the contents.

Parameters **text** (*str*) – text to add

contextMenuEvent(*self*, *event*)
Reimplemented method to add a clear action into the default context menu.

Parameters **event** (*QContextMenuEvent*) – Received event

widgets.custom_qtreeview

Classes for custom QTreeView.

author

M. Marin (KTH)

date 25.4.2018

Module Contents

class widgets.custom_qtreeview.**CopyTreeView**(*parent*)
Bases: PySide2.QtWidgets.QTreeView

Custom QTreeView class with copy support.

copy(*self*)
Copy current selection to clipboard in excel format.

class widgets.custom_qtreeview.**ObjectTreeView**(*parent*)
Bases: *widgets.custom_qtreeview.CopyTreeView*

Custom QTreeView class for object tree in TreeViewForm.

parent
The parent of this view

Type QWidget

edit_key_pressed

edit(*self*, *index*, *trigger*, *event*)
Send signal instead of editing item, so the TreeViewForm can catch this signal and open a custom QDialog for edition.

class widgets.custom_qtreeview.**ReferencesTreeView**(*parent*)
Bases: PySide2.QtWidgets.QTreeView

Custom QTreeView class for 'References' in Data Connection properties.

parent
The parent of this view

Type QWidget

files_dropped

del_key_pressed

dragEnterEvent(*self*, *event*)
Accept file drops from the filesystem.

dragMoveEvent(*self*, *event*)
Accept event.

dropEvent(*self*, *event*)
Emit files_dropped signal with a list of files for each dropped url.

keyPressEvent(*self*, *event*)
Overridden method to make the view support deleting items with a delete key.

class widgets.custom_qtreeview.**DataTreeView**(*parent*)
Bases: PySide2.QtWidgets.QTreeView

Custom QTreeView class for 'Data' in Data Connection properties.

parent
The parent of this view

Type QWidget

files_dropped

del_key_pressed

dragEnterEvent(*self*, *event*)
Accept file drops from the filesystem.

dragMoveEvent(*self*, *event*)
Accept event.

dropEvent(*self*, *event*)
Emit files_dropped signal with a list of files for each dropped url.

mousePressEvent(*self*, *event*)
Register drag start position.

mouseMoveEvent(*self*, *event*)
Start dragging action if needed.

mouseReleaseEvent(*self*, *event*)
Forget drag start position

keyPressEvent(*self*, *event*)
Overridden method to make the view support deleting items with a delete key.

class widgets.custom_qtreeview.**SourcesTreeView**(*parent*)
Bases: PySide2.QtWidgets.QTreeView

Custom QTreeView class for 'Sources' in Tool Template form.

parent
The parent of this view

Type QWidget

files_dropped

del_key_pressed

dragEnterEvent(*self*, *event*)
Accept file and folder drops from the filesystem.

dragMoveEvent(*self*, *event*)
Accept event.

dropEvent(*self*, *event*)
Emit files_dropped signal with a list of files for each dropped url.

keyPressEvent(*self*, *event*)
Overridden method to make the view support deleting items with a delete key.

class widgets.custom_qtreeview.**CustomTreeView**(*parent*)
Bases: PySide2.QtWidgets.QTreeView

Custom QTreeView class for Tool template editor form to enable keyPressEvent.

parent
The parent of this view

Type QWidget

del_key_pressed

keyPressEvent(*self*, *event*)
Overridden method to make the view support deleting items with a delete key.

widgets.custom_qwidgets

Custom QWidgets for Filtering and Zooming.

author

P. Vennström (VTT)

date 4.12.2018

Module Contents

class widgets.custom_qwidgets.**FilterWidget**(*parent=None*, *show_empty=True*)
Bases: PySide2.QtWidgets.QWidget

Filter widget class.

okPressed

cancelPressed

save_state(*self*)
Saves the state of the FilterCheckboxListModel.

reset_state(*self*)
Sets the state of the FilterCheckboxListModel to saved state.

clear_filter(*self*)
Selects all items in FilterCheckBoxListModel.

has_filter(*self*)
Returns true if any item is filtered in FilterCheckboxListModel false otherwise.

set_filter_list(*self*, *data*)
Sets the list of items to filter.

_apply_filter(*self*)
Apply current filter and save state.

_cancel_filter(*self*)
Cancel current edit of filter and set the state to the stored state.

_filter_list(*self*)
Filter list with current text.

_text_edited(*self*, *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 widgets.custom_qwidgets.**ZoomWidget**(*parent=None*)
Bases: PySide2.QtWidgets.QWidget

A widget for a QWidgetAction providing zoom actions for a graph view.

Attributes parent (QWidget): the widget's parent

minus_pressed

plus_pressed

reset_pressed

paintEvent(*self*, *event*)
Overridden method.

widgets.data_store_widget

Contains the DataStoreForm class, parent class of TreeViewForm and GraphViewForm.

author

M. Marin (KTH)

date 26.11.2018

Module Contents

class widgets.data_store_widget.**DataStoreForm**(*project*, *ui*, *db_maps*)
Bases: PySide2.QtWidgets.QMainWindow

A widget to show and edit Spine objects in a data store.

project
The project instance that owns this form

Type *SpineToolboxProject*

ui
UI definition of the form that is initialized

db_maps
named DiffDatabaseMapping instances

Type dict

msg

msg_error

commit_available

add_toggle_view_actions(*self*)
Add toggle view actions to View menu.

connect_signals(*self*)
Connect signals to slots.

qsettings(*self*)
Returns the QSettings instance from ToolboxUI.

add_message(*self*, *msg*)
Append regular message to status bar.

Parameters **msg** (*str*) – String to show in QStatusBar

add_error_message(*self*, *msg*)
Show error message.

Parameters **msg** (*str*) – String to show in QErrorMessage

_handle_object_parameter_value_visibility_changed(*self*, *visible*)

_handle_object_parameter_definition_visibility_changed(*self*, *visible*)

_handle_relationship_parameter_value_visibility_changed(*self*, *visible*)

_handle_relationship_parameter_definition_visibility_changed(*self*, *visible*)

_handle_tag_button_toggled(*self*, *db_map_ids*, *checked*)
Called when a parameter tag button is toggled. Compute selected parameter definition ids per object class ids. Then update set of selected object class ids. Finally, update filter.

_handle_commit_available(*self*, *on*)

show_commit_session_dialog(*self*, *checked=False*)
Query user for a commit message and commit changes to source database.

commit_session(*self*, *commit_msg*)

rollback_session(*self*, *checked=False*)

refresh_session(*self*, *checked=False*)

init_models(*self*)
Initialize models.

init_object_tree_model(*self*)
Initialize object tree model.

init_parameter_value_models(*self*)
Initialize parameter value models from source database.

init_parameter_definition_models(*self*)
Initialize parameter (definition) models from source database.

init_parameter_value_list_model(*self*)
Initialize parameter value_list models from source database.

init_parameter_tag_toolbar(*self*)
Initialize parameter tag toolbar.

setup_delegates(*self*)
Set delegates for tables.

all_selected_object_class_ids(*self*)
Return object class ids selected in object tree *and* parameter tag toolbar.

all_selected_relationship_class_ids(*self*)
Return relationship class ids selected in relationship tree *and* parameter tag toolbar.

set_default_parameter_rows(*self*, *index=None*)
Set default rows for parameter models according to selection in object or relationship tree.

do_update_filter(*self*)
Apply filter on visible views.

show_add_object_classes_form(*self*, *checked=False*)
Show dialog to let user select preferences for new object classes.

show_add_objects_form(*self*, *checked=False*, *class_name=''*)
Show dialog to let user select preferences for new objects.

show_add_relationship_classes_form(*self*, *checked=False*, *object_class_one_name=None*)
Show dialog to let user select preferences for new relationship class.

show_add_relationships_form(*self*, *checked=False*, *relationship_class_key=()*, *object_class_name=''*, *object_name=''*)
Show dialog to let user select preferences for new relationships.

add_object_classes(*self*, *object_class_d*)
Insert new object classes.

add_object_classses_to_models(*self*, *db_map*, *added*)

add_objects(*self*, *object_d*)
Insert new objects.

add_relationship_classes(*self*, *rel_cls_d*)
Insert new relationship classes.

add_relationship_classes_to_models(*self*, *db_map*, *added*)

add_relationships(*self*, *relationship_d*)
Insert new relationships.

add_relationships_to_models(*self*, *db_map*, *added*)

show_edit_object_classes_form(*self*, *checked=False*)

show_edit_objects_form(*self*, *checked=False*)

show_edit_relationship_classes_form(*self*, *checked=False*)

show_edit_relationships_form(*self*, *checked=False*)

update_object_classes(*self*, *object_class_d*)
Update object classes.

update_object_classes_in_models(*self*, *db_map*, *updated*)

update_objects(*self*, *object_d*)
Update objects.

update_objects_in_models(*self*, *db_map*, *updated*)

update_relationship_classes(*self*, *rel_cls_d*)
Update relationship classes.

update_relationship_classes_in_models(*self*, *db_map*, *updated*)

update_relationships (*self*, *relationship_d*)
Update relationships.

update_relationships_in_models (*self*, *db_map*, *updated*)

add_parameter_value_lists (*self*, *parameter_value_list_d*)

update_parameter_value_lists (*self*, *parameter_value_list_d*)

show_manage_parameter_tags_form (*self*, *checked=False*)

add_parameter_tags (*self*, *parameter_tag_d*)
Add parameter tags.

update_parameter_tags (*self*, *parameter_tag_d*)
Update parameter tags.

remove_parameter_tags (*self*, *parameter_tag_d*)
Remove parameter tags.

show_parameter_value_editor (*self*, *index*, *table_view*, *value=None*)
Shows the parameter value editor for the given index of given table view.

set_parameter_value_data (*self*, *index*, *new_value*)
Update (object or relationship) parameter value with newly edited data.

set_parameter_definition_data (*self*, *index*, *new_value*)
Update (object or relationship) parameter definition with newly edited data. If the parameter name changed, update it in (object or relationship) parameter value.

show_commit_session_prompt (*self*)
Shows the commit session message box.

restore_ui (*self*)
Restore UI state from previous session.

closeEvent (*self*, *event=None*)
Handle close window.

Parameters **event** (*QEvent*) – Closing event if 'X' is clicked.

widgets.datetime_editor

An editor widget for editing datetime database (relationship) parameter values.

author

A. Soininen (VTT)

date 28.6.2019

Module Contents

widgets.datetime_editor.**_QDateTime_to_datetime** (*dt*)
Converts a QDateTime object to Python's datetime.datetime type.

widgets.datetime_editor.**_datetime_to_QDateTime** (*dt*)
Converts Python's datetime.datetime object to QDateTime.

class 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(*self*, *new_datetime*)
Updates the internal DateTime value

set_value(*self*, *value*)
Sets the value to be edited.

value(*self*)
Returns the editor's current value.

widgets.duration_editor

An editor widget for editing duration database (relationship) parameter values.

author

A. Soininen (VTT)

date 28.6.2019

Module Contents

widgets.duration_editor.**_to_text**(*value*)
Converts a Duration object to a string of comma separated time durations.

class 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(*self*)
Updates the value being edited.

set_value(*self*, *value*)
Sets the value for editing.

value(*self*)
Returns the current Duration.

widgets.graph_view_widget

Contains the TreeViewForm class.

author

M. Marin (KTH)

date 26.11.2018

Module Contents

class widgets.graph_view_widget.**GraphViewForm**(*project*, *db_maps*, *read_only=False*)
Bases: *widgets.data_store_widget.DataStoreForm*

A widget to show the graph view.

project
The project instance that owns this form

Type *SpineToolboxProject*

db_maps
named DiffDatabaseMapping instances

Type dict

read_only
Whether or not the form should be editable

Type bool

show(*self*)
Show usage message together with the form.

init_models(*self*)
Initialize models.

init_parameter_value_models(*self*)
Initialize parameter value models from source database.

init_parameter_definition_models(*self*)
Initialize parameter (definition) models from source database.

setup_zoom_action(*self*)
Setup zoom action in view menu.

create_add_more_actions(*self*)
Create and 'Add more' action and button for the Item Palette views.

connect_signals(*self*)
Connect signals.

restore_dock_widgets(*self*)
Dock all floating and or hidden QDockWidgets back to the window at 'factory' positions.

_handle_zoom_widget_minus_pressed(*self*)

_handle_zoom_widget_plus_pressed(*self*)

_handle_zoom_widget_reset_pressed(*self*)

_handle_zoom_widget_action_hovered(*self*)
Called when the zoom widget action is hovered. Hide the 'Dock widgets' submenu in case it's being shown. This is the default behavior for hovering 'normal' 'QAction's, but for some reason it's not the case for hovering 'QWidgetAction's.

_handle_menu_about_to_show(*self*)
Called when a menu from the menubar is about to show.

_handle_item_palette_dock_location_changed(*self*, *area*)
Called when the item palette dock widget location changes. Adjust splitter orientation accordingly.

add_toggle_view_actions(*self*)
Add toggle view actions to View menu.

init_commit_rollback_actions(*self*)

build_graph(*self*, *checked=True*)
Initialize graph data and build graph.

_handle_object_tree_selection_changed(*self*, *selected*, *deselected*)
Build_graph.

init_graph_data(*self*)
Initialize graph data.

static shortest_path_matrix(*object_name_list*, *src_ind_list*, *dst_ind_list*, *spread*)
Return the shortest-path matrix.

static sets(*N*)
Return sets of vertex pairs indices.

static vertex_coordinates(*matrix*, *heavy_positions=None*, *iterations=10*, *weight_exp=-2*, *initial_diameter=1000*)
Return x and y coordinates for each vertex in the graph, computed using VSGD-MS.

make_graph(*self*)
Make graph.

new_scene(*self*)
A new scene with a background.

extend_scene(*self*)
Make scene rect the size of the scene show all items.

_handle_scene_selection_changed(*self*)
Show parameters for selected items.

_handle_scene_changed(*self*, *region*)
Handle scene changed. Show usage message if no items other than the bg.

show_usage_msg(*self*)
Show usage instructions in new scene.

_handle_usage_link_activated(*self*, *link*)

_handle_item_dropped(*self*, *pos*, *text*)

relationship_items(*self*, *object_name_list*, *object_class_name_list*, *extent*, *spread*, *label_color*, *object_class_id_list=None*, *relationship_class_id=None*)
Lists of object and arc items that form a relationship.

add_relationship_template(*self*, *scene*, *x*, *y*, *object_items*, *arc_items*, *dimension_at_origin=None*)
Add relationship parts into the scene to form a 'relationship template'.

add_object(*self*, *object_item*, *name*)
Try and add object given an object item and a name.

update_object(*self*, *object_item*, *name*)
Try and update object given an object item and a name.

add_relationship(*self*, *template_id*, *object_items*)
Try and add relationship given a template id and a list of object items.

add_object_classses_to_models (*self*, *db_map*, *added*)

add_relationship_classes_to_models (*self*, *db_map*, *added*)
Insert new relationship classes.

show_graph_view_context_menu (*self*, *global_pos*)
Show context menu for graphics view.

hide_selected_items (*self*, *checked=False*)
Hide selected items.

show_hidden_items (*self*, *checked=False*)
Show hidden items.

prune_selected_items (*self*, *checked=False*)
Prune selected items.

reinstate_pruned_items (*self*, *checked=False*)
Reinstate pruned items.

show_object_item_context_menu (*self*, *e*, *main_item*)
Show context menu for object_item.

show_object_parameter_value_context_menu (*self*, *pos*)

show_object_parameter_definition_context_menu (*self*, *pos*)

show_relationship_parameter_value_context_menu (*self*, *pos*)

show_relationship_parameter_definition_context_menu (*self*, *pos*)

_show_table_context_menu (*self*, *position*, *table_view*, *column_name*)

remove_graph_items (*self*, *checked=False*)
Remove all selected items in the graph.

closeEvent (*self*, *event=None*)
Handle close window.

Parameters **event** (*QEvent*) – Closing event if 'X' is clicked.

widgets.import_errors_widget

Contains ImportErrorWidget class.

author

P. Vennström (VTT)

date 1.6.2019

Module Contents

class widgets.import_errors_widget.**ImportErrorWidget** (*parent=None*)
Bases: PySide2.QtWidgets.QWidget

Widget to display errors while importing and ask user for action.

set_import_state (*self*, *num_imported*, *errors*)
Sets state of error widget.

Parameters

- **{int} -- number of successfully imported items** (*num_imported*) –
- **{list} -- list of errors.** (*errors*) –

widgets.import_preview_widget

Contains ImportPreviewWidget, and MappingTableMenu classes.

author

P. Vennström (VTT)

date 1.6.2019

Module Contents

class widgets.import_preview_widget.**ImportPreviewWidget**(*connector*, *parent=None*)
Bases: PySide2.QtWidgets.QWidget

A Widget for defining one or more Mappings associated to a data Source (CSV file, Excel file, etc). Currently it's being embedded in ImportDialog and ImportPreviewWindow.

Parameters **connector** (ConnectionManager) –

tableChecked

mappedDataReady

previewDataUpdated

checked_tables

set_loading_status(*self*, *status*)
Sets widgets enable state

connection_ready(*self*)
Requests new tables data from connector

select_table(*self*, *selection*)
Set selected table and request data from connector

check_list_item(*self*, *item*)
Set the check state of item

handle_connector_error(*self*, *error_message*)

request_mapped_data(*self*)

update_tables(*self*, *tables*)
Update list of tables

update_preview_data(*self*, *data*, *header*)

use_settings(*self*, *settings*)

get_settings_dict(*self*)
Returns a dictionary with type of connector, connector options for tables, mappings for tables, selected tables.

Returns [Dict] – dict with settings

close_connection(*self*)
close connector connection

class widgets.import_preview_widget.**MappingTableMenu**(*parent=None*)
Bases: PySide2.QtWidgets.QMenu

A menu to let users define a Mapping from a data table. Used to generate the context menu for ImportPreviewWidget._ui_table

set_model(*self*, *model*)

set_mapping(*self*, *name=''*, *map_type=None*, *value=None*)

ignore_columns(*self*, *columns=None*)

request_menu(*self*, *QPos=None*)

widgets.import_preview_window

Contains DataInterface class.

authors
P. Savolainen (VTT)

date 10.6.2019

Module Contents

class widgets.import_preview_window.**ImportPreviewWindow**(*data_interface*, *filepath*, *connector*, *settings*)
Bases: PySide2.QtWidgets.QMainWindow

A QMainWindow to let users define Mappings for a Data Interface item.

settings_updated

connection_failed

save(*self*)

save_and_close(*self*)

start_ui(*self*)

restore_ui(*self*)
Restore UI state from previous session.

closeEvent(*self*, *event=None*)
Handle close window.

Parameters **event** (*QEvent*) – Closing event if 'X' is clicked.

widgets.import_widget

ImportDialog class.

author
P. Vennström (VTT)

date 1.6.2019

Module Contents

class widgets.import_widget.**ImportDialog**(*parent=None*)
Bases: PySide2.QtWidgets.QDialog

A widget for importing data into a Spine db. Currently used by TreeViewForm. It embeds three widgets that alternate depending on user's actions: - *select_widget* is a *QWidget* for selecting the source data type (CSV, Excel, etc.) - *_import_preview* is an *ImportPreviewWidget* for defining Mappings to associate with the source data - *_error_widget* is an *ImportErrorWidget* to show errors from import operations

mapped_data

mapping_errors

connector_selected(*self*, *selection*)

set_ok_button_availability(*self*)

import_data(*self*, *data*, *errors*)

data_ready(*self*, *data*, *errors*)

ok_clicked(*self*)

cancel_clicked(*self*)

back_clicked(*self*)

launch_import_preview(*self*)

_handle_failed_connection(*self*, *msg*)
Handle failed connection, show error message and select widget

Parameters **{str} -- str with message of reason for failed connection.** (*msg*) –

set_preview_as_main_widget(*self*)

set_error_widget_as_main_widget(*self*)

widgets.import_widget.**app**

widgets.indexed_value_table_context_menu

Offers a convenience function for time pattern and time series editor widgets.

author

A. Soininen (VTT)

date 5.7.2019

Module Contents

widgets.indexed_value_table_context_menu.**handle_table_context_menu**(*click_pos*, *table_view*, *model*, *parent_widget*)
Shows a context menu for parameter value tables and handles the selection.

Parameters

- **{QPoint)** (*click_pos*) – position from the context menu event
- **table_view** (*QTableView*) – the table widget
- **model** (TimePatternModel, TimeSeriesModelFixedResolution, TimeSeriesModelVariableResolution) – a model
- **(QWidget** (*parent_widget*) – context menu's parent widget

widgets.indexed_value_table_context_menu.**_remove_rows**(*selected_rows*, *model*)
Packs consecutive rows into a single removeRows call.

widgets.julia_repl_widget

Class for a custom RichJupyterWidget to use as julia REPL.

author

M. Marin (KTH)

date 22.5.2018

Module Contents

class widgets.julia_repl_widget.**CustomQtKernelManager**
Bases: qtconsole.manager.QtKernelManager

A QtKernelManager with a custom restarter, and a means to override the –project argument.

kernel_left_dead

project_path

kernel_spec

override_project_arg(*self*)

start_restarter(*self*)
Start a restarter with custom time to dead and restart limit.

_handle_kernel_left_dead(*self*)

class widgets.julia_repl_widget.**JuliaREPLWidget**(*toolbox*)
Bases: qtconsole.rich_jupyter_widget.RichJupyterWidget

Class for a custom RichJupyterWidget.

toolbox
QMainWindow instance

Type *ToolboxUI*

execution_finished_signal

julia_kernel_name(*self*)
Returns the name of the julia kernel specification, according to the selected julia interpreter in settings. Returns None if julia version cannot be determined.

start_jupyter_kernel(*self*)
Start a Julia Jupyter kernel if available.

Returns True if the kernel is started, or in process of being started (installing/reconfiguring IJulia) False if the kernel cannot be started and the user chooses not to install/reconfigure IJulia

start_available_jupyter_kernel(*self*)
Start a Jupyter kernel which is available (from the attribute *kernel_name*)

Returns True if the kernel is started, or in process of being started (reconfiguring IJulia) False if the kernel cannot be started and the user chooses not to reconfigure IJulia

check_ijulia(*self*)
Check if IJulia is installed, returns True, False, or None if unable to determine.

handle_repl_failed_to_start(*self*)
Prompt user to install IJulia if missing, or rebuild it otherwise.

Returns Boolean value depending on whether or not the problem is being handled.

restart_jupyter_kernel(*self*)
Restart the julia jupyter kernel if it's already started. Otherwise, or if the julia version has changed in settings, start a new jupyter kernel.

setup_client(*self*)

_handle_kernel_restarted(*self*, *died=True*)
Called when the kernel is restarted, i.e., when time to dead has elapsed.

_handle_kernel_left_dead(*self*)
Called when the kernel is finally declared dead, i.e., the restart limit has been reached.

handle_ijulia_installation_finished(*self*, *ret*)
Run when IJulia installation process finishes

handle_ijulia_rebuild_finished(*self*, *ret*)
Run when IJulia rebuild process finishes

check_ijulia_process(*self*, *ret*)
Check whether or not the IJulia process finished successfully

_handle_execute_reply(*self*, *msg*)

_handle_status(*self*, *msg*)
Handle status message. If we have a command in line and the kernel reports status 'idle', execute that command.

_handle_error(*self*, *msg*)
Handle error messages.

execute_instance(*self*, *command*)
Try and start the jupyter kernel. Execute command immediately if kernel is idle. If not, it will be executed as soon as the kernel becomes idle (see *_handle_status* method).

terminate_process(*self*)
Send interrupt signal to kernel.

shutdown_jupyter_kernel(*self*)
Shut down the jupyter kernel.

_context_menu_make(*self*, *pos*)
Reimplemented to add an action for (re)start REPL action.

enterEvent(*self*, *event*)
Set busy cursor during REPL (re)starts.

dragEnterEvent (*self*, *e*)
Don't accept drops from Add Item Toolbar.

copy_input (*self*)
Copy only input.

widgets.mapping_widget

MappingWidget and MappingOptionsWidget class.

author

P. Vennström (VTT)

date 1.6.2019

Module Contents

widgets.mapping_widget.**MAPPING_CHOICES = ['Constant', 'Column', 'Row', 'Header', 'None']**

class widgets.mapping_widget.**MappingWidget** (*parent=None*)
Bases: PySide2.QtWidgets.QWidget

A widget for managing Mappings (add, remove, edit, visualize, and so on). Intended to be embeded in a ImportPreviewWidget.

mappingChanged

mappingDataChanged

set_data_source_column_num (*self*, *num*)

set_model (*self*, *model*)
Sets new model

data_changed (*self*)

new_mapping (*self*)
Adds new empty mapping

delete_selected_mapping (*self*)
deletes selected mapping

select_mapping (*self*, *selection*)
gets selected mapping and emits mappingChanged

class widgets.mapping_widget.**MappingOptionsWidget** (*parent=None*)
Bases: PySide2.QtWidgets.QWidget

A widget for managing Mapping options (class type, dimensions, parameter type, ignore columns, and so on). Intended to be embeded in a MappingWidget.

set_num_available_columns (*self*, *num*)

change_skip_columns (*self*, *filterw*, *skip_cols*, *has_filter*)

set_model (*self*, *model*)

update_ui (*self*)
updates ui to RelationshipClassMapping or ObjectClassMapping model

change_class (*self*, *new_class*)

change_dimension(*self*, *dim*)

change_parameter(*self*, *par*)

change_import_objects(*self*, *state*)

widgets.options_widget

Contains OptionsWidget class.

author

P. Vennström (VTT)

date 1.6.2019

Module Contents

class widgets.options_widget.**OptionsWidget**(*options*, *header='Options'*, *parent=None*)
Bases: PySide2.QtWidgets.QWidget

A widget for handling simple options. Used by ConnectionManager.

optionsChanged

_build_ui(*self*)
Builds ui from specification in dict

set_options(*self*, *options=None*, *set_missing_default=True*)
Sets state of options

Keyword Arguments

- **{Dict} -- Dict with option name as key and value as value (default** (*options*) – {None})
- **{bool} -- Sets missing options to default if True (default** (*set_missing_default*) – {True})

get_options(*self*)
Returns current state of option widget

Returns [Dict] – Dict with option name as key and value as value

widgets.parameter_value_editor

An editor dialog for editing database (relationship) parameter values.

author

A. Soininen (VTT)

date 28.6.2019

Module Contents

class widgets.parameter_value_editor.**_Editor**
Bases: enum.Enum

Indexes for the specialized editors corresponding to the selector combo box and editor stack.

PLAIN_VALUE = 0

TIME_SERIES_FIXED_RESOLUTION = 1

TIME_SERIES_VARIABLE_RESOLUTION = 2

TIME_PATTERN = 3

DATETIME = 4

DURATION = 5

class widgets.parameter_value_editor.**ParameterValueEditor**(*parent_index*, *value_name=''*, *value=None*, *parent_widget=None*)

Bases: PySide2.QtWidgets.QDialog

Dialog for editing (relationship) parameter values.

The dialog takes the editable value from a parent model 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 parent model.

parent_index
an index to a parameter value in parent_model

Type QModelIndex

value_name
name of the value

Type str

value
parameter value or None if it should be loaded from parent_index

parent_widget
a parent widget

Type QWidget

accept(*self*)
Saves the parameter value shown in the currently selected editor widget back to the parent model.

_change_parameter_type(*self*, *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(*self*, *value*)
Shows the editor widget corresponding to the given value type on the editor stack.

_warn_and_select_default_view(*self*, *message*)
Displays a warning dialog and opens the default editor widget after user clicks OK.

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

class widgets.plain_parameter_value_editor.**_ValueModel**(*value*)
A model to handle the parameter value in the editor.

Mostly useful because of the handy conversion of strings to floats or booleans.

value
a parameter value

Type float, bool

value
Returns the value held by the model.

class 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_value(*self*, *value*)
Sets the value to be edited in this widget.

_value_changed(*self*)
Updates the model.

value(*self*)
Returns the value currently being edited.

widgets.plot_canvas

A Qt widget to use as a matplotlib backend.

author

A. Soininen (VTT)

date 3.6.2019

Module Contents

widgets.plot_canvas.**_mpl_logger**

class widgets.plot_canvas.**PlotCanvas**(*parent=None*)
Bases: matplotlib.backends.backend_qt5agg.FigureCanvasQTAgg

A widget for plotting with matplotlib

widgets.plot_widget

A Qt widget showing a toolbar and a matplotlib plotting canvas.

author

A. Soininen (VTT)

date 27.6.2019

Module Contents

class widgets.plot_widget.**PlotWidget**(*parent=None*)
Bases: PySide2.QtWidgets.QWidget

widgets.project_form_widget

Widget shown to user when a new project is created.

authors

P. Savolainen (VTT)

date 10.1.2018

Module Contents

class widgets.project_form_widget.**NewProjectForm**(*toolbox*)
Bases: PySide2.QtWidgets.QWidget

Class for a new project widget.

toolbox
Parent widget.

Type *ToolboxUI*

connect_signals(*self*)
Connect signals to slots.

name_changed(*self*)
Update label to show a preview of the project directory name.

ok_clicked(*self*)
Check that project name is valid and create project.

call_create_project(*self*)
Call ToolboxUI method create_project().

keyPressEvent(*self*, *e*)
Close project form when escape key is pressed.

Parameters **e** (*QKeyEvent*) – Received key press event.

closeEvent(*self*, *event=None*)
Handle close window.

Parameters **event** (*QEvent*) – Closing event if 'X' is clicked.

widgets.python_repl_widget

Class for a custom RichJupyterWidget to use as Python REPL.

author

P. Savolainen (VTT)

date 14.3.2019

Module Contents

class widgets.python_repl_widget.**PythonReplWidget**(*toolbox*)
Bases: qtconsole.rich_jupyter_widget.RichJupyterWidget

Python Repl Widget class.

toolbox
App main window (QMainWindow) instance

Type *ToolboxUI*

execution_finished_signal

connect_signals(*self*)
Connect signals.

disconnect_signals(*self*)
Disconnect signals. Needed before switching to another Python kernel.

python_kernel_name(*self*)
Returns the name of the Python kernel specification and its display name according to the selected Python environment in Settings. Returns None if Python version cannot be determined.

setup_python_kernel(*self*)
Context menu Start action handler.

launch_kernel(*self*, *k_name*, *k_display_name*)
Check if selected kernel exists or if it needs to be set up before launching.

check_and_install_requirements(*self*)
Prompts user to install IPython and ipykernel if they are missing. After installing the required packages, installs kernelspecs for the selected Python if they are missing.

Returns Boolean value depending on whether or not the user chooses to proceed.

is_package_installed(*self*, *package_name*)
Checks if given package is installed to selected Python environment.

Parameters **package_name** (*str*) – Package name

Returns True if installed, False if not

Return type (bool)

start_package_install_process(*self*, *package_name*)
Starts installing the given package using pip.

Parameters **package_name** (*str*) – Package name to install using pip

package_install_process_finished(*self*, *retval*)
Installing package finished.

Parameters **retval** (*int*) – Process return value. 0: success, !0: failure

start_kernelspec_install_process(*self*)
Install kernel specifications for the selected Python environment.

kernelspec_install_process_finished(*self*, *retval*)
Installing package finished.

Parameters **retval** (*int*) – Process return value. 0: success, !0: failure

start_python_kernel(*self*)
Starts kernel manager and client and attaches the client to the Python Console.

execute_instance(*self*, *commands*)
Start executing the first command in the command queue in Python Console.

execution_in_progress(*self*, *code*)
Slot for handling the 'executing' signal. Signal is emitted when a user visible 'execute_request' has been submitted to the kernel from the FrontendWidget.

Parameters **code** (*str*) – Code to be executed (actually not 'str' but 'object')

execution_done(*self*, *msg*)
Slot for handling the 'executed' signal. Signal is emitted when a user-visible 'execute_reply' has been received from the kernel and processed by the FrontendWidget.

Parameters **msg** (*dict*) – Response message (actually not 'dict' but 'object')

iopub_msg_received(*self*, *msg*)
Message received from the IOPUB channel. Note: We are only monitoring when the kernel has started successfully and ready for action here. Alternatively, this could be done in the Slot for the 'executed' signal. However, this Slot could come in handy at some point. See 'Messaging in Jupyter' for details: https://jupyter-client.readthedocs.io/en/latest/messaging.html

Parameters **msg** (*dict*) – Received message from IOPUB channel

terminate_process(*self*)
Send interrupt signal to kernel.

shutdown_kernel(*self*, *hush=False*)
Shut down Python kernel.

push_vars(*self*, *var_name*, *var_value*)
Push a variable to Python Console session. Simply executes command 'var_name=var_value'.

Parameters

- **var_name** (*str*) – Variable name
- **var_value** (*object*) – Variable value

Returns True if succeeded, False otherwise

Return type (bool)

test_push_vars(*self*)
QAction slot to test pushing variables to Python Console.

_context_menu_make(*self*, *pos*)
Reimplemented to add custom actions.

dragEnterEvent(*self*, *e*)
Don't accept project item drops.

widgets.report_plotting_failure

Functions to report failures in plotting to the user.

author

A. Soininen (VTT)

date 10.7.2019

Module Contents

widgets.report_plotting_failure.**report_plotting_failure**(*error*)
Reports a PlottingError exception to the user.

widgets.settings_widget

Widget for controlling user settings.

author

P. Savolainen (VTT)

date 17.1.2018

Module Contents

class widgets.settings_widget.**SettingsWidget**(*toolbox*)
Bases: PySide2.QtWidgets.QWidget

A widget to change user's preferred settings.

toolbox
Parent widget.

Type *ToolboxUI*

connect_signals(*self*)
Connect signals.

browse_gams_path(*self*, *checked=False*)
Open file browser where user can select a GAMS program.

browse_julia_path(*self*, *checked=False*)
Open file browser where user can select a Julia executable (i.e. julia.exe on Windows).

browse_julia_project_path(*self*, *checked=False*)
Open file browser where user can select a Julia project path.

browse_python_path(*self*, *checked=False*)
Open file browser where user can select a python interpreter (i.e. python.exe on Windows).

browse_work_path(*self*, *checked=False*)
Open file browser where user can select the path to wanted work directory.

show_color_dialog(*self*, *checked=False*)
Let user pick the bg color.

Parameters **checked** (*boolean*) – Value emitted with clicked signal

update_bg_color(*self*)
Set tool button icon as the selected color and update Design View scene background color.

update_scene_bg(*self*, *checked*)
Draw background on scene depending on radiobutton states.

Parameters **checked** (*boolean*) – Toggle state

read_settings(*self*)
Read saved settings from app QSettings instance and update UI to display them.

read_project_settings(*self*)
Read project settings from config object and update settings widgets accordingly.

ok_clicked(*self*)
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.

update_project_settings(*self*)
Update project settings when Ok has been clicked.

check_if_python_env_changed(*self*, *new_path*)
Checks if Python environment was changed. This indicates that the Python Console may need a restart.

file_is_valid(*self*, *file_path*, *msgbox_title*)
Checks that given path is not a directory and it's a file that actually exists. Needed because the QLineEdits are editable.

dir_is_valid(*self*, *dir_path*, *msgbox_title*)
Checks that given path is a directory. Needed because the QLineEdits are editable.

keyPressEvent(*self*, *e*)
Close settings form when escape key is pressed.

Parameters **e** (*QKeyEvent*) – Received key press event.

closeEvent(*self*, *event=None*)
Handle close window.

Parameters **event** (*QEvent*) – Closing event if 'X' is clicked.

mousePressEvent(*self*, *e*)
Save mouse position at the start of dragging.

Parameters **e** (*QMouseEvent*) – Mouse event

mouseReleaseEvent(*self*, *e*)
Save mouse position at the end of dragging.

Parameters **e** (*QMouseEvent*) – Mouse event

mouseMoveEvent(*self*, *e*)
Moves the window when mouse button is pressed and mouse cursor is moved.

Parameters **e** (*QMouseEvent*) – Mouse event

widgets.spine_datapackage_widget

Widget shown to user when opening a 'datapackage.json' file in Data Connection item.

author

M. Marin (KTH)

date 7.7.2018

Module Contents

class widgets.spine_datapackage_widget.**SpineDatapackageWidget**(*data_connection*)
Bases: PySide2.QtWidgets.QMainWindow

A widget to allow user to edit a datapackage and convert it to a Spine database in SQLite.

data_connection
Data Connection associated to this widget

Type *DataConnection*

msg

msg_proc

msg_error

add_toggle_view_actions(*self*)
Add toggle view actions to View menu.

show(*self*)
Called when the form shows. Init datapackage (either from existing datapackage.json or by inferring a new one from sources) and update ui.

infer_datapackage(*self*, *checked=False*)
Called when the user triggers the infer action. Infer datapackage from sources and update ui.

load_datapackage(*self*)
Load datapackage from 'datapackage.json' file in data directory, or infer one from CSV files in that directory.

infer_datapackage_(*self*)
Infer datapackage from CSV files in data directory.

update_ui(*self*)
Update ui from datapackage attribute.

connect_signals(*self*)
Connect signals to slots.

restore_ui(*self*)
Restore UI state from previous session.

_handle_menu_about_to_show(*self*)
Called when a menu from the menubar is about to show. Adjust infer action depending on whether or not we have a datapackage. Adjust copy paste actions depending on which widget has the focus. TODO Enable/disable action to save datapackage depending on status.

add_message(*self*, *msg*)
Prepend regular message to status bar.

Parameters **msg** (*str*) – String to show in QStatusBar

add_process_message(*self*, *msg*)
Show process message in status bar. This messages stays until replaced.

Parameters **msg** (*str*) – String to show in QStatusBar

add_error_message(*self*, *msg*)
Show error message.

Parameters **msg** (*str*) – String to show

save_datapackage(*self*, *checked=False*)
Write datapackage to file 'datapackage.json' in data directory.

show_export_to_spine_dialog(*self*, *checked=False*)
Show dialog to allow user to select a file to export.

export_to_spine(*self*, *file_path*)
Export datapackage into Spine SQlite file.

_handle_converter_progressed(*self*, *step*, *msg*)

_handle_converter_failed(*self*, *msg*)

_handle_converter_finished(*self*)

copy(*self*, *checked=False*)
Copy data to clipboard.

paste(*self*, *checked=False*)
Paste data from clipboard.

load_resource_data(*self*)
Load resource data into a local list of tables.

reset_resource_models(*self*, *current*, *previous*)
Reset resource data and schema models whenever a new resource is selected.

reset_resource_data_model(*self*)
Reset resource data model with data from newly selected resource.

update_resource_data(*self*, *index*, *new_value*)
Update resource data with newly edited data.

_handle_resource_name_data_committed(*self*, *index*, *new_name*)
Called when line edit delegate wants to edit resource name data. Update resources model and descriptor with new resource name.

_handle_field_name_data_committed(*self*, *index*, *new_name*)
Called when line edit delegate wants to edit field name data. Update name in fields_model, resource_data_model's header and datapackage descriptor.

_handle_primary_key_data_committed(*self*, *index*)
Called when checkbox delegate wants to edit primary key data. Add or remove primary key field accordingly.

_handle_foreign_keys_data_committed(*self*, *index*, *value*)

_handle_foreign_keys_data_changed(*self*, *top_left*, *bottom_right*, *roles=None*)
Called when foreign keys data is updated in model. Update descriptor accordingly.

_handle_foreign_keys_model_rows_inserted(*self*, *parent*, *first*, *last*)

create_remove_foreign_keys_row_button(*self*, *index*)
Create button to remove foreign keys row.

remove_foreign_key_row(*self*, *button*)

closeEvent(*self*, *event=None*)
Handle close event.

Parameters **event** (*QEvent*) – Closing event if 'X' is clicked.

class widgets.spine_datapackage_widget.**CustomPackage**(*descriptor=None*, *base_path=None*, *strict=False*, *storage=None*)

Bases: datapackage.Package

Custom datapackage class.

rename_resource(*self*, *old*, *new*)

rename_field(*self*, *resource*, *old*, *new*)
Rename a field.

set_primary_key(*self*, *resource*, **primary_key*)
Set primary key for a given resource in the package

append_to_primary_key(*self*, *resource*, *field*)
Append field to resources's primary key.

remove_from_primary_key(*self*, *resource*, *field*)
Remove field from resources's primary key.

insert_foreign_key(*self*, *row*, *resource_name*, *field_names*, *reference_resource_name*, *reference_field_names*)
Insert foreign key to a given resource in the package at a given row.

remove_primary_key(*self*, *resource*, **primary_key*)
Remove the primary key for a given resource in the package

remove_foreign_key(*self*, *resource*, *fields*, *reference_resource*, *reference_fields*)
Remove foreign key from the package

remove_foreign_keys_row(*self*, *row*, *resource*)
Remove foreign keys row from the package

widgets.tabular_view_widget

Contains TabularViewForm class and some related constants.

author

P. Vennström (VTT)

date 1.11.2018

Module Contents

widgets.tabular_view_widget.**ParameterValue**

widgets.tabular_view_widget.**RELATIONSHIP_CLASS = relationship**

widgets.tabular_view_widget.**OBJECT_CLASS = object**

widgets.tabular_view_widget.**DATA_JSON = json**

widgets.tabular_view_widget.**DATA_VALUE = value**

widgets.tabular_view_widget.**DATA_SET = set**

widgets.tabular_view_widget.**JSON_TIME_NAME = json time**

widgets.tabular_view_widget.**PARAMETER_NAME = db parameter**

widgets.tabular_view_widget.**unpack_json**(*data*)

class widgets.tabular_view_widget.**TabularViewForm**(*data_store*, *db_map*, *database*)

Bases: PySide2.QtWidgets.QMainWindow

A widget to show and edit Spine objects in a data store.

data_store
The DataStore instance that owns this form

Type *DataStore*

db_map
The object relational database mapping

Type DatabaseMapping

database
The database name

Type str

pivot_table_edit(*self*, *index*, *trigger*, *event*)
Starts editing the item at index from pivot_table. If the index contains some 'complex' parameter value, we open the parameter value editor window instead.

set_session_menu_enable(*self*)
Checks if session can commit or rollback and updates session menu actions

load_class_data(*self*)

load_objects(*self*)

load_relationships(*self*)

load_parameter_values(*self*)

current_object_class_list(*self*)

get_set_data(*self*)

update_class_list(*self*)
update list_select_class with all object classes and relationship classes

show_commit_session_dialog(*self*)
Query user for a commit message and commit changes to source database.

commit_session(*self*, *commit_msg*)

rollback_session(*self*)

model_has_changes(*self*)
checks if PivotModel has any changes

change_frozen_value(*self*, *newSelection*)

get_selected_class(*self*)

pack_dict_json(*self*)
Pack down values with json_index into a json_array

delete_parameter_values(*self*, *delete_values*)

delete_relationships(*self*, *delete_relationships*)

delete_index_values_from_db(*self*, *delete_indexes*)

add_index_values_to_db(*self*, *add_indexes*)

save_model_set(*self*)

save_model(*self*)

save_parameter_values(*self*, *data*, *data_value*)

save_relationships(*self*)

update_pivot_lists_to_new_model(*self*)

update_frozen_table_to_model(*self*)

change_class(*self*)

get_pivot_preferences(*self*, *selection_key*, *index_names*)

get_valid_entries_dicts(*self*)

select_data(*self*, *text=''*)

table_index_entries_changed(*self*, *added_entries*, *deleted_enties*)

update_filters_to_new_model(*self*)

create_filter_widget(*self*, *name*)

change_filter(*self*, *menu*, *valid*, *has_filter*)

change_pivot(*self*, *parent*, *event*)

find_frozen_values(*self*, *frozen*)

show_commit_session_prompt(*self*)
Shows the commit session message box.

restore_ui(*self*)
Restore UI state from previous session.

save_ui(*self*)
Saves UI state

closeEvent(*self*, *event=None*)
Handle close window.

Parameters **event** (*QEvent*) – Closing event if 'X' is clicked.

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

class widgets.time_pattern_editor.**TimePatternEditor**(*parent=None*)
Bases: PySide2.QtWidgets.QWidget

A widget for editing time patterns.

parent

Type QWidget

_show_table_context_menu(*self*, *pos*)

set_value(*self*, *value*)
Sets the parameter value to be edited.

value(*self*)
Returns the parameter value currently being edited.

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

widgets.time_series_fixed_resolution_editor.**_resolution_to_text**(*resolution*)
Converts a list of durations into a string of comma-separated durations.

widgets.time_series_fixed_resolution_editor.**_text_to_resolution**(*text*)
Converts a comma-separated string of durations into a resolution array.

class widgets.time_series_fixed_resolution_editor.**TimeSeriesFixedResolutionEditor**(*parent=Non*
Bases: PySide2.QtWidgets.QWidget

A widget for editing time series data with a fixed time step.

parent
a parent widget

Type QWidget

_resolution_changed(*self*)
Updates the models after resolution change.

_show_table_context_menu(*self*, *pos*)
Shows the table's context menu.

_select_date(*self*, *selected_date*)

set_value(*self*, *value*)
Sets the parameter value for editing in this widget.

_show_calendar(*self*)

_start_time_changed(*self*)
Updates the model due to start time change.

_update_plot(*self*, *topLeft=None*, *bottomRight=None*, *roles=None*)
Updated the plot.

value(*self*)
Returns the parameter value currently being edited.

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

class widgets.time_series_variable_resolution_editor.**TimeSeriesVariableResolutionEditor**(*par*

Bases: PySide2.QtWidgets.QWidget

A widget for editing variable resolution time series data.

parent
a parent widget

Type QWidget

_show_table_context_menu(*self*, *pos*)
Shows the table's context menu.

set_value(*self*, *value*)
Sets the time series being edited.

_update_plot(*self*, *topLeft=None*, *bottomRight=None*, *roles=None*)
Updates the plot widget.

value(*self*)
Return the time series currently being edited.

widgets.tool_configuration_assistant_widget

Widget for assisting the user in configuring tools, such as SpineModel.

author

M. Marin (KTH)

date 9.1.2019

Module Contents

class widgets.tool_configuration_assistant_widget.**ToolConfigurationAssistantWidget**(*toolbox*, *au-*
torun=True

Bases: PySide2.QtWidgets.QWidget

A widget to assist the user in configuring external tools such as SpineModel.

toolbox
Parent widget.

Type *ToolboxUI*

autorun
whether or not to start configuration process at form load

Type bool

connect_signals(*self*)
Connect signals.

add_spine_model_msg(*self*, *msg*)
Append message to SpineModel log.

Parameters **msg** (*str*) – String written to QTextBrowser

add_spine_model_error_msg(*self*, *msg*)
Append error message to SpineModel log.

Parameters **msg** (*str*) – String written to QTextBrowser

add_spine_model_success_msg(*self*, *msg*)
Append success message to SpineModel log.

Parameters **msg** (*str*) – String written to QTextBrowser

configure_spine_model(*self*)
Run when form loads. Check SpineModel version.

_handle_spine_model_version_check_finished(*self*, *ret*)
Run when the Spine Model configuration assistant has finished checking SpineModel version. Install SpineModel if not found, otherwise check the python program used by PyCall.

_handle_spine_model_installation_finished(*self*, *ret*)
Run when the Spine Model configuration assistant has finished installing SpineModel. Check the python program used by PyCall.

_handle_py_call_program_check_finished(*self*, *ret*)
Run when the Spine Model configuration assistant has finished checking the python program used by PyCall. Install PyCall if not found, otherwise reconfigure PyCall to use same python as Spine Toolbox if it's not the case.

_handle_py_call_installation_finished(*self*, *ret*)
Run when the Spine Model configuration assistant has finished installing PyCall. Check the python program used by PyCall.

_handle_py_call_reconfiguration_finished(*self*, *ret*)
Run when the Spine Model configuration assistant has finished reconfiguring PyCall. End Spine Model configuration.

get_permission(*self*, *title*, *action*)
Ask user's permission to perform an action and return True if granted.

closeEvent(*self*, *event=None*)
Handle close widget.

Parameters **event** (*QEvent*) – PySide2 event

widgets.tool_template_widget

QWidget that is used to create or edit Tool Templates. In the former case it is presented empty, but in the latter it is filled with all the information from the template being edited.

author

M. Marin (KTH), P. Savolainen (VTT)

date 12.4.2018

Module Contents

class widgets.tool_template_widget.**ToolTemplateWidget**(*toolbox*, *tool_template=None*)

Bases: PySide2.QtWidgets.QWidget

A widget to query user's preferences for a new tool template.

toolbox
QMainWindow instance

Type *ToolboxUI*

tool_template
If given, the form is pre-filled with this template

Type *ToolTemplate*

connect_signals(*self*)
Connect signals to slots.

populate_sourcefile_list(*self*, *items*)
List source files in QTreeView. If items is None or empty list, model is cleared.

populate_inputfiles_list(*self*, *items*)
List input files in QTreeView. If items is None or empty list, model is cleared.

populate_inputfiles_opt_list(*self*, *items*)
List optional input files in QTreeView. If items is None or empty list, model is cleared.

populate_outputfiles_list(*self*, *items*)
List output files in QTreeView. If items is None or empty list, model is cleared.

browse_main_program(*self*, *checked=False*)
Open file browser where user can select the path of the main program file.

set_main_program_path(*self*, *file_path*)
Set main program file and folder path.

new_main_program_file(*self*)
Create a new blank main program file. Let user decide the file name and location.

new_main_program_file(*self*)
Creates a new blank main program file. Let's user decide the file name and path. Alternative version using only one getSaveFileName dialog.

new_source_file(*self*)
Let user create a new source file for this tool template.

show_add_source_files_dialog(*self*, *checked=False*)
Let user select source files for this tool template.

show_add_source_dirs_dialog(*self*, *checked=False*)
Let user select a source directory for this tool template. All files and sub-directories will be added to the source files.

add_dropped_includes(*self*, *file_paths*)

add_single_include(*self*, *path*)
Add file path to Source files list.

open_includes_file(*self*, *index*)
Open source file in default program.

remove_source_files_with_del (*self*)
Support for deleting items with the Delete key.

remove_source_files (*self*, *checked=False*)
Remove selected source files from include list. Do not remove anything if there are no items selected.

add_inputfiles (*self*, *checked=False*)
Let user select input files for this tool template.

remove_inputfiles_with_del (*self*)
Support for deleting items with the Delete key.

remove_inputfiles (*self*, *checked=False*)
Remove selected input files from list. Do not remove anything if there are no items selected.

add_inputfiles_opt (*self*, *checked=False*)
Let user select optional input files for this tool template.

remove_inputfiles_opt_with_del (*self*)
Support for deleting items with the Delete key.

remove_inputfiles_opt (*self*, *checked=False*)
Remove selected optional input files from list. Do not remove anything if there are no items selected.

add_outputfiles (*self*, *checked=False*)
Let user select output files for this tool template.

remove_outputfiles_with_del (*self*)
Support for deleting items with the Delete key.

remove_outputfiles (*self*, *checked=False*)
Remove selected output files from list. Do not remove anything if there are no items selected.

ok_clicked (*self*)
Check that everything is valid, create definition dictionary and add template to project.

call_add_tool_template (*self*)
Add or update Tool Template according to user's selections. If the name is the same as an existing tool template, it is updated and auto-saved to the definition file. (User is editing an existing tool template.) If the name is not in the tool template model, create a new tool template and offer to save the definition file. (User is creating a new tool template from scratch or spawning from an existing one).

keyPressEvent (*self*, *e*)
Close Setup form when escape key is pressed.

Parameters **e** (*QKeyEvent*) – Received key press event.

closeEvent (*self*, *event=None*)
Handle close window.

Parameters **event** (*QEvent*) – Closing event if 'X' is clicked.

widgets.toolbars

Functions to make and handle QToolBars.

author

P. Savolainen (VTT)

date 19.1.2018

Module Contents

class widgets.toolbars.**ItemToolBar**(*parent*)
Bases: PySide2.QtWidgets.QToolBar

A toolbar to add items using drag and drop actions.

parent
QMainWindow instance

Type *ToolboxUI*

remove_all(*self*, *checked=False*)
Slot for handling the remove all tool button clicked signal. Calls ToolboxUI remove_all_items() method.

execute_project(*self*, *checked=False*)
Slot for handling the Execute project tool button clicked signal.

execute_selected(*self*, *checked=False*)
Slot for handling the Execute selected tool button clicked signal.

stop_execution(*self*, *checked=False*)
Slot for handling the Stop execution tool button clicked signal.

class widgets.toolbars.**DraggableWidget**(*parent*, *pixmap*, *text*)
Bases: PySide2.QtWidgets.QLabel

A draggable QLabel.

parent
Parent widget

Type QWidget

pixmap
Picture for the label

Type QPixMap

text
Item type

Type str

mousePressEvent(*self*, *event*)
Register drag start position

mouseMoveEvent(*self*, *event*)
Start dragging action if needed

mouseReleaseEvent(*self*, *event*)
Forget drag start position

class widgets.toolbars.**ParameterTagToolBar**(*parent*)
Bases: PySide2.QtWidgets.QToolBar

A toolbar to add items using drag and drop actions.

parent
tree or graph view form

Type *DataStoreForm*

tag_button_toggled

manage_tags_action_triggered

init_toolbar(*self*)

add_tag_actions(*self*, *db_map*, *parameter_tags*)

remove_tag_actions(*self*, *db_map*, *parameter_tag_ids*)

update_tag_actions(*self*, *db_map*, *parameter_tags*)

widgets.tree_view_widget

Contains the TreeViewForm class.

author

M. Marin (KTH)

date 26.11.2018

Module Contents

class widgets.tree_view_widget.**TreeViewForm**(*project*, *db_maps*)
Bases: *widgets.data_store_widget.DataStoreForm*

A widget to show and edit Spine objects in a data store.

project
The project instance that owns this form

Type *SpineToolboxProject*

db_maps
named DiffDatabaseMapping instances

Type dict

object_class_selection_available

object_selection_available

relationship_class_selection_available

relationship_selection_available

object_tree_selection_available

relationship_tree_selection_available

obj_parameter_definition_selection_available

obj_parameter_value_selection_available

rel_parameter_definition_selection_available

rel_parameter_value_selection_available

parameter_value_list_selection_available

add_toggle_view_actions(*self*)
Add toggle view actions to View menu.

connect_signals(*self*)
Connect signals to slots.

restore_dock_widgets(*self*)
Dock all floating and or hidden QDockWidgets back to the window at 'factory' positions.

update_copy_and_remove_actions(*self*)
Update copy and remove actions according to selections across the widgets.

_handle_object_tree_selection_available(*self*, *on*)

_handle_relationship_tree_selection_available(*self*, *on*)

_handle_obj_parameter_definition_selection_available(*self*, *on*)

_handle_obj_parameter_value_selection_available(*self*, *on*)

_handle_rel_parameter_definition_selection_available(*self*, *on*)

_handle_rel_parameter_value_selection_available(*self*, *on*)

_handle_parameter_value_list_selection_available(*self*, *on*)

update_paste_action(*self*, *old*, *new*)

copy(*self*, *checked=False*)
Copy data to clipboard.

paste(*self*, *checked=False*)
Paste data from clipboard.

remove_selection(*self*, *checked=False*)
Remove selection of items.

_handle_object_parameter_definition_selection_changed(*self*, *selected*, *deselected*)
Enable/disable the option to remove rows.

_handle_object_parameter_value_selection_changed(*self*, *selected*, *deselected*)
Enable/disable the option to remove rows.

_handle_relationship_parameter_definition_selection_changed(*self*, *selected*, *deselected*)
Enable/disable the option to remove rows.

_handle_relationship_parameter_value_selection_changed(*self*, *selected*, *deselected*)
Enable/disable the option to remove rows.

_handle_parameter_value_list_selection_changed(*self*, *selected*, *deselected*)
Enable/disable the option to remove rows.

_handle_object_parameter_tab_changed(*self*, *index*)
Update filter.

_handle_relationship_parameter_tab_changed(*self*, *index*)
Update filter.

show_import_file_dialog(*self*, *checked=False*)
Show dialog to allow user to select a file to import.

import_file(*self*, *file_path*, *checked=False*)
Import data from file into current database.

export_database(*self*, *checked=False*)
Exports a database to a file.

_select_database(*self*)
Lets user select a database from available databases.

Shows a dialog from which user can select a single database. If there is only a single database it is selected automatically and no dialog is shown.

Returns the database map of the database or None if no database was selected

export_to_excel(*self*, *db_map*, *file_path*)
Export data from database into Excel file.

export_to_sqlite(*self*, *db_map*, *file_path*)
Export data from database into SQlite file.

init_models(*self*)
Initialize models.

init_object_tree_model(*self*)
Initialize object tree model.

init_relationship_tree_model(*self*)
Initialize relationship tree model.

find_next_leaf(*self*, *index*)
If object tree index corresponds to a relationship, then expand the next ocurrence of it.

find_next(*self*, *index*)
Expand next occurrence of a relationship in object tree.

clear_other_selections(*self*, **skip_widgets*)
Clear selections in all widgets except *skip_widgets*.

_handle_object_tree_selection_changed(*self*, *selected*, *deselected*)
Called when the object tree selection changes. Set default rows and apply filters on parameter models.

_handle_relationship_tree_selection_changed(*self*, *selected*, *deselected*)
Called when the relationship tree selection changes. Set default rows and apply filters on parameter models.

update_filter(*self*)
Update filters on parameter models according to selected and deselected object tree indexes.

show_object_tree_context_menu(*self*, *pos*)
Context menu for object tree.

Parameters **pos** (*QPoint*) – Mouse position

show_relationship_tree_context_menu(*self*, *pos*)
Context menu for relationship tree.

Parameters **pos** (*QPoint*) – Mouse position

fully_expand_selection(*self*)

fully_collapse_selection(*self*)

call_show_add_objects_form(*self*, *index*)

call_show_add_relationship_classes_form(*self*, *index*)

call_show_add_relationships_form(*self*, *index*)

add_object_classes(*self*, *object_class_d*)
Insert new object classes.

add_relationship_classes_to_models(*self*, *db_map*, *added*)

add_relationships_to_models(*self*, *db_map*, *added*)

edit_object_tree_items(*self*)
Called when F2 is pressed while the object tree has focus. Call the appropriate method to show the edit form, depending on the current index.

edit_relationship_tree_items(*self*)
Called when F2 is pressed while the relationship tree has focus. Call the appropriate method to show the edit form, depending on the current index.

update_object_classes_in_models(*self*, *db_map*, *updated*)

update_objects_in_models(*self*, *db_map*, *updated*)

update_relationship_classes_in_models(*self*, *db_map*, *updated*)

update_relationships_in_models(*self*, *db_map*, *updated*)

show_remove_object_tree_items_form(*self*)
Show form to remove items from object treeview.

show_remove_relationship_tree_items_form(*self*)
Show form to remove items from relationship treeview.

remove_tree_items(*self*, *item_d*)
Remove items from tree views.

show_object_parameter_value_context_menu(*self*, *pos*)
Context menu for object parameter value table view.

Parameters **pos** (*QPoint*) – Mouse position

show_relationship_parameter_value_context_menu(*self*, *pos*)
Context menu for relationship parameter value table view.

Parameters **pos** (*QPoint*) – Mouse position

show_object_parameter_context_menu(*self*, *pos*)
Context menu for object parameter table view.

Parameters **pos** (*QPoint*) – Mouse position

show_relationship_parameter_context_menu(*self*, *pos*)
Context menu for relationship parameter table view.

Parameters **pos** (*QPoint*) – Mouse position

_show_parameter_context_menu(*self*, *position*, *table_view*, *value_column_header*, *remove_selection*)
Show a context menu for parameter tables.

Parameters

- **position** (*QPoint*) – local mouse position in the table view
- **table_view** (*QTableView*) – the table view where the context menu was triggered
- **value_column_header** (*str*) – column header for editable/plottable values

show_parameter_value_list_context_menu(*self*, *pos*)
Context menu for relationship parameter table view.

Parameters **pos** (*QPoint*) – Mouse position

remove_object_parameter_values(*self*)
Remove selected rows from object parameter value table.

remove_relationship_parameter_values(*self*)
Remove selected rows from relationship parameter value table.

_remove_parameter_values(*self*, *table_view*)
Remove selected rows from parameter value table.

Parameters **table_view** (*QTableView*) – a table view from which to remove

remove_object_parameter_definitions(*self*)
Remove selected rows from object parameter definition table.

remove_relationship_parameter_definitions(*self*)
Remove selected rows from relationship parameter definition table.

_remove_parameter_definitions(*self*, *table_view*, *value_model*, *class_id_header*)
Remove selected rows from parameter table.

Parameters

- **table_view** (*QTableView*) – the table widget from which to remove
- **value_model** (*QAbstractTableModel*) – a value model corresponding to the definition model of table_view
- **class_id_header** (*str*) – header of the class id column

remove_parameter_value_lists(*self*)
Remove selection of parameter value_lists.

CHAPTER 17

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z