Spine Toolbox Documentation

Release 0.8.0-final.0

Spine project consortium

Apr 30, 2024

CONTENTS:

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

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

CHAPTER

ONE

WHAT'S NEW?

Here's the Changelog for Spine Toolbox.

```
# Changelog
All **notable** changes to this project are documented here.

The format is based on [Keep a Changelog](http://keepachangelog.com/en/1.0.0/)

## [Unreleased]

### Added

### Changed

#### Miscellaneous changes

### Deprecated

### Removed

### Fixed

### Security

## 0.8.0

### Added

- New context menu action (Select superclass) for entity class items in the entity tree.
- Added Tool Specification type (Python, Gams, etc.) icons on Design View.
- There is now a new filter type, Alternative filter available in Link properties.
  Unlike scenario filters, the execution is not parallelized.
  Instead, a successor item sees parameter values of all selected alternatives.
  Because of this behavior,
  alternative filters cannot be used at the same time with scenario filters.
  Link properties tab has a combo box that lets one choose which filter type to use.

### Changed

#### Spine data structure
```

(continues on next page)

(continued from previous page)

```
Many parts of the Spine data structure have been redesigned.

- *Entities* have replaced objects and relationships.
  Zero-dimensional entities correspond to objects while multidimensional entities␣
↪replace the former relationships.
  Unlike relationships, the *elements* of multidimensional entities can now be other␣
↪multidimensional entities.
- Simple inheritance is now supported by *superclasses*.
- Tools, features and methods have been removed.
  The functionality that was previously implemented using the is_active parameter
  has been replaced by *entity alternatives*.
  Entity classes have a default setting for the entity alternative called *active by␣
↪default*.
  Database migration should automatically replace tools, features and methods
  by entity alternatives and set active by default to whatever default value `is_active`
  or similar parameter had.
  The `is_active` parameter is not removed from entity classes but its values are.
- Note that new zero-dimensional entity classes have *active by default* set to `false`␣
↪initially.
  This means that the entities of those classes are hidden when using scenario filters
  unless specifically shown using entity alternatives.

#### Miscellaneous changes

- "Rubber band" selection of items in Design and Graph views is now done with **left␣
↪mouse button**
  (no need to press Ctrl anymore). The views can be dragged around by holding the␣
↪**right mouse button**.
- Spine Database Editor now remembers the configuration of the docs in each view for a␣
↪specific URL. The docks
  can be reset from the hamburger menu **View->Docks...->Reset docks**.
- You can now select a different Julia executable & project or Julia kernel for each␣
↪Tool spec.
  This overrides the global setting from Toolbox Settings.
- Headless mode now supports remote execution (see 'python -m spinetoolbox --help')
- Commit Viewer's UI has undergone some redesigning and can now handle large databases.

### Removed

- Project dock widget
- Dependency on Dagster

## [0.7.4]

### Changed

- Microsoft build tools are not needed anymore when installing Toolbox requirements on␣
↪Python 3.10 and up.

## [0.7.3]

### Added
```

(continues on next page)

(continued from previous page)

```
- On Toolbox startup, a link appears in Event log that opens Upgrade notification window
  offering information about the upcoming 0.8 update.

## [0.7.2] - 2023-12-04

### Added

- Data Connection items now support schemas in database references
- Importer Specification Editor now supports database schemas

## [0.7.1] - 2023-10-06

### Added

- Allow choosing of highlight color for items found in entity graph

### Fixed

- Fixed issues when remote databases were used as references in Data Connections
- Fixed a bug where the order of scenario alternatives would be messed up when copy-
↪pasting
  the alternatives in the Scenario Tree
- Fixed a bug with time series plots
- Fixed issues with Pivot table

## [0.7.0] - 2023-08-25

### Added
- Support for version 11 Spine Toolbox projects.
- Executable Tool Specifications can be used to run any (shell) command. This enhancement
  duplicates the functionality of Gimlet project items and makes them obsolete.
- There is now an option to select if new scenarios or tools are automatically used
  for filtering in Link properties.
- The new "Filer validation" button in Link properties allows forcing at least one␣
↪scenario or tool
  filter to be checked.
- Python 3.11 support.
- PySide6 support. The app has been ported from PySide2 to PySide6.
- In addition to dragging and dropping,
  it is now possible to copy alternatives from Alternative tree and paste them to the␣
↪scenario tree
  in Database editor.
- Scenarios can now be copied and pasted between databases in Database editor.
- Scenarios can now be duplicated in Database editor.
- Tool project item's Python and Julia consoles can now be killed from the right-click␣
↪context menu.
  A killed console can be restored by restarting it.
- A new option "Kill consoles at the end of execution" has been added to Tool properties,
  that kills Python and Julia console processes after execution saving some memory and␣
↪computing resources.
- A new option "Log process output to a file" has been added to Tool properties,
```

(continues on next page)

(continued from previous page)

```
  that logs all the 'console' output of the Tool to a file in the logs subfolder.
- You can now open a 'Detached' Jupyter Console on any kernel you have installed in your␣
↪system from the *Consoles*
  main window menu.
- "Make Julia Kernel" and "Make Python Kernel" buttons in Settings->Tools page. Clicking␣
↪them creates a new
  Julia or Python kernel based on selected Julia/Python executable on the same page if␣
↪the kernel does not exist.
  If the kernel already exists, it is selected automatically.
- ``project.json`` now has an experimental option ["project"]["settings"]["enable_
↪execute_all"] which disables the
  Execute Project button when set to ``false``. The option is currently not settable in␣
↪the UI.
- New context menu action (Find...) to find items by name in DB editor's entity graph.

### Changed
- The console settings of Python tools as well as the command and shell settings of␣
↪executable tools
  are now treated as project-specific data
  and stored in ``<project root>/.spinetoolbox/local/specification_local_data.json``.
  This should make it easier to share projects over e.g. Git.
- Scenario and tool filters are now ON by default.
  If new scenario or tool is added to a database it will be automatically selected in␣
↪outgoing connections.
  NOTE: In existing projects, if a connection did not have any scenarios/tools selected,
  they will all be selected when opening a project for the first time after this change.
  If this is not desired, the scenarios/tools need to be deselected manually
  before saving the project on disk.
- Project name is now the project directory name and cannot be changed unless by moving␣
↪the project to another
  directory manually or by using the File -> Save project as... menu item.
- Scenario filters now filter unrelated scenarios and alternatives as well.
- Alternative/Scenario tree in Database editor has been split into separate Alternative␣
↪tree and Scenario tree docks.
  This will hopefully make dragging and dropping alternatives to scenarios easier.
- Logic of executing selected project items. URL's passed by unselected Data␣
↪Transformers are not included automatically
  into selective execution anymore. When selecting project items for execution, make␣
↪sure to select also preceding
  Data Transformers if needed.
- Names for duplicate items/scenarios/etc now use the format `prefix (xx)` instead of␣
↪`prefix xx`.
  Duplicating a previously duplicated item now has the number `xx` incremented instead␣
↪of having a new number appended.
- "Open kernel spec editor" buttons in Settings->Tools page have been changed "Make␣
↪Julia kernel" and
  "Make Python Kernel" buttons
-

### Deprecated

### Removed
```

(continues on next page)

(continued from previous page)

```
- Python 3.7 support
- GdxExporter project item. Please use Exporter instead.
- Gimlet project item. Please use Tool instead.
- The possibility to import parameter values when importing object groups using Importer.
  Existing mappings will not import group parameters anymore.
  Please add explicit parameter import mappings if needed.
- The possibility to export parameter values when exporting object groups using Exporter.
  Existing mappings will not export group parameters anymore.
  Please add explicit parameter export mappings if needed.
- Execute in work setting in Tool Specification Editor. Please use Execute in work␣
↪setting
  in Tool Properties instead.
- Kernel Spec Editor widget and the "Open Kernel Spec Editor" button in Tool␣
↪Specification Editor

### Fixed
- Deleting the last item of a mapping in the importer specification editor no longer␣
↪disables the 'add' button.
- Group Id's for Jupyter Consoles
- Kill consoles at end of execution for Jupyter Consoles
- Crash when typing exit, exit(), quit, or quit() into Jupyter Console

### Security

## [0.6.5] - 2021-09-08

### Added
- Support for loops has been added. Loops can be created using a special *loop link*␣
↪which is initiated by
  holding down the **Alt** key while drawing a new link in Design view.
- Performance boost for Spine Db Editor when working with large databases.

### Changed
- Installation does not require cx-Oracle and psycopg2 packages anymore.
- install_requires and requirements.txt files have been revised.

### Fixed
- Spine Db Editor Graph View works on Python 3.7 again.
- Spine Db Editor Graph View 'Add objects' and 'Save position' actions.
- Export to SQLite in Spine Db Editor

## [0.6.3] - 2021-09-03

### Added
- Plenty of stability improvements and bug fixes

### Fixed
- Fixed 'ImportError: DLL load failed while importing win32api' on (Conda) Python 3.8␣
↪when trying to execute Tools
  in Jupyter Console

## [0.6.1] - 2021-06-28
```

(continues on next page)

(continued from previous page)

```
### Added
- Data Transformer now supports parameter value transformations.
- Project execution shortcuts: F5 to execute all DAGs, F6 to execute selected items
  and F7 to stop execution.
- Time series, maps and other compound values have gained the ability to have names for␣
↪their indexes.
  Index names can be edited in parameter value editors, and they are also supported by␣
↪Importer and Exporter items.
- Support for running Python Tools (specifications) in a Conda environment
- Execution mode (kernel spec, console, interpreter) can now be selected individually␣
↪for each Python
  Tool specification

### Changed
- Data Transformer's specification editor has now a new interface.
- Parameter renaming in Data Transformer requires now entity class names to identify the␣
↪parameters.
  Data Transformer's icon will show a notification if class names are missing.
- Installation instructions advice to install directly from PyPI.
- Stand-alone DB Editor is now opened with the `spine-db-editor [URL]` command
- Python settings on the *Tools* page of *File->Settings* are now the default settings␣
↪for new Python Tool
  specifications. I.e. they are not global settings anymore.

### Deprecated
- GdxExporter has been deprecated. Use the general purpose Exporter item instead.
  GdxExporter will be removed in a future release. Please replace existing items by␣
↪Exporter.

## [0.6.0-final.2] - 2021-06-03

### Fixed
- [win-x64] Running Python or Julia Tools does not open an extra console window anymore

### Security
- urllib3 v1.26.5 now required because of a security vulnerability in earlier versions

## [0.6.0-final.1] - 2021-06-01

### Fixed
- Event Log and Item Execution Logs now automatically scroll to the bottom when there␣
↪are new messages
- [win-x64] Resolve correct GAMS, Python, and Julia paths in Settings->Tools

## [0.6.0-final.0] - 2021-05-07

### Added
- Support for parallel/multicore processing
- New project item: Data Transformer. Can be used to configure Spine database filters␣
↪for successor items.
  Currently, it supports renaming entity classes.
```

(continues on next page)

(continued from previous page)

```
- New project item: Exporter. A general-purpose tabular data exporter.
- Support for version 3 Spine Toolbox projects and an automatic upgrade of version 2␣
↪projects to version 3.
- Support for version 4 Spine Toolbox projects.
- Support for version 5 Spine Toolbox projects.
- Support for version 6 Spine Toolbox projects.
- Support to create sysimages for Julia tools.
- New requirement: jill, for installing Julia.
- The SpineOpt configuration assistant has been moved from File->Configuration␣
↪assistants,
  to File->Settings->Tools->Julia, and renamed to SpineOpt Installer.
- New wizard to install Julia, accessible from File->Settings->Tools->Julia.
- File->Close project option
- Support for Python 3.8
- Automated kernel creation, if the user selects to run tools in console without having␣
↪created a kernel.
- Option to pack CSV resource files into one datapackage.json file for advertising,␣
↪available from Link properties.
- Option to color project item icons in the toolbar, available frm File->Settings->
↪General.
- Reorganize project item icons in the toolbar with drag and drop.

### Changed
- Project Item (Tool, Data Store, Importer, etc.) code has been removed from Spine␣
↪Toolbox.
  Project Items are now in a separate package called spine_items, which is upgraded at␣
↪Spine
  Toolbox's startup.
- Importer item now applies the same mapping to all input files. If the user needs to␣
↪apply different
  mappings, they need to create different Importers. The specification can be shared␣
↪using the json file.
- The .gdx exporter project item is now called GdxExporter.
- [win-x64] Installer does not require admin rights anymore
- [win-x64] Installer always asks for an installation directory, even if a previous␣
↪installation exists
- [win-x64] Installer wizard style changed to modern

### Removed
- Combiner project item. The same functionality can be achieved by connecting a Data␣
↪Store to another Data Store.
- Upgrade support for original (.proj file based) Spine Toolbox projects.
- Python 3.6 is no longer supported.
- The Spine Datapackage Editor is gone. There wasn't enough reason to keep this widget
- The app no longer checks that Spine dependencies are up to date. Users are asked to␣
↪follow the upgrade procedure
  which involves manually upgrading requirements after pulling the latest master branch.

### Fixed
- [win-x64] returning_process.py when frozen
- Traceback in GdxExporter when there are indexing settings for a parameter that is not␣
↪in the database
```

(continues on next page)

(continued from previous page)

```
- Bug in installing Plugins
- Traceback when removing Plugins

## [0.5.0-final.1] - 2020-02-03

### Added
- Tutorial for case study A5 in the documentation

### Fixed
- [win-x64] Fixed /tools/python.exe by adding sitecustomize.py and a missing python37.dll

## [0.5.0-final.0] - 2020-12-14

### Added
- Exporting graphs as PDF files from the *Graph* menu in the Data Store form.
- Pruning entire classes from the graph view. The option is available both from the␣
↪*Graph* menu and
  from *Entity Graph* context menus. Also, pruned items can be restored gradually.
- A new Input type *Indexed parameter expansion* is now available in Data Store view's␣
↪Pivot table.
  In this Input type the indexes, e.g. time stamps of time series get expanded as a new␣
↪dimension in the table.
- Import editor now has a new Source type: Table name. It can be used e.g. to pick an␣
↪Excel sheet's
  or GAMS domain's name as the object class name.
- Import editor now supports multidimensional maps. The number of dimensions can be set␣
↪using the
  *Map dimensions* spin box in mappings options.
- Executing a project from the command line without opening the Toolbox GUI (i.e.␣
↪headless execution).
  The headless execution is enabled by the new command line option ``--execute-only``.
- Toolbox now supports scenarios and alternatives. They can be accessed via Data store␣
↪view's new Alternative tree.
- New Project Item: Gimlet. Can be used to run any command as part of the workflow
  with or without a shell. Supported shells at the moment are cmd and powershell for
  Windows and bash for other OS's.
- Python and Julia Kernel spec Editor. Provides the means to make new kernel specs for␣
↪Python Console and Julia
  Console without leaving Spine Toolbox. Kernel (spec) Editor can be found in Settings->
↪Tools tab.
- [win-x64] Includes Tools/python.exe for running Python Tools for systems that do not␣
↪have a Python installation.
  Also, pyvenv.cfg and path.pth files for configuring the included python.exe.

### Fixed
- Signal disconnection issue in Graph View
- Bugs in removing objects and object classes in Spine db editor's Graph View

### Changed
- Data Store Form is now called 'Spine database editor'
- Spine db editor Graph View behavior. Now selecting objects in the object tree not only␣
↪shows those objects but also
```

(continues on next page)

(continued from previous page)

```
  all the cascading relationships. One can still go back to the previous behavior in␣
↪Settings.
- Moving object items in the graph view also causes relationship icons to follow. This␣
↪behavior can be disabled in the
  Settings.
- Required PySide2 version is now 5.14. The version is checked at startup.
- Indexed parameter handling has been overhauled in Exporter allowing parameters to␣
↪share indexing domains.
  **Note**: Due to numerous changes in the backend, Exporters in old project files will␣
↪not load properly
  and need to be re-configured.
- The way Exporter handles missing parameter values and None values has changed. The␣
↪item now ignores missing
  values instead of replacing them by the default value. Further, there is a new option␣
↪to replace None values by
  the default value and another option to replace Nones by not-a-numbers or skip␣
↪exporting them.
- The numerical indicator on the upper left corner of project items no longer indicates␣
↪the execution order for
  each individual item because the exact order is not know before the Execute button is␣
↪actually clicked.
  The number still indicates the execution order but may show the same numbers for items␣
↪in different parallel
  branches.
- Project.json file format has been upgraded to version 2. Version 1 project.json files␣
↪are upgraded to version 2
  automatically when a project is opened.
- Default Python interpreter is now {sys.executable} i.e. the one that was used in␣
↪launching the app.
  This affects the Python used by Python Tool specifications and the PyCall used by␣
↪SpineOpt.jl configuration
  assistant.
- [win-x64] Default Python interpreter is the Python in user's PATH if available. If␣
↪Python is not defined in
  user's PATH, the default Python interpreter is the <app_install_dir>/Tools/python.exe.
- User's now need to explicitly choose the kernel specs for the Python Console and the␣
↪Julia Console. They are
  not chosen (nor created) automatically anymore. The kernel specs can be selected in␣
↪the drop-down menus
  in application Settings->Tools.
- Database revision handling has been improved. Id est, the app does not offer to␣
↪upgrade databases
  that are more recent than the current version of Spine Toolbox can handle.
- Links to User Guide and Getting Started documents open only the online versions. The␣
↪docs have been
  published on readthedocs.org.
- Clearing the line edits for Julia executable and Python Interpreter (in Settings->
↪Tools) shows
  the full paths to their respective files as placeholder text.

### Deprecated
- CustomQtKernelManager class
```

(continues on next page)

(continued from previous page)

```
### Removed
- python_repl_widget.py
- julia_repl_widget.py

## [0.4.0-final.0] - 2020-04-03

### Added
- A small notification icon is painted next to project items in the design view whenever they
  are missing some configuration. Hovering the icon shows tips for completing the
  configuration.
- A small icon is painted next to the project items in the design view to show the order in
  which they will be executed
- Main Window menu 'File -> Open recent'. Shortcut for opening a recent project.
- A new project item *Exporter* allows a database contained in a *Data Store* to be exported
  as GAMS `.gdx` file.
- It is now possible to copy and paste project items for example between projects.
- It is now possible to duplicate project items.
- Changes made in the tree view are also seen in the graph view and viceversa.
- New Setting: *Sticky selection in Graph View*. Enables users to select if they want to use
  multi-selection or single selection in the Graph view Object tree when selecting items with
  the **left-mouse button**.
- Projects can be saved to any directory
- Project name can be changed in Settings
- The graph view features a short live demonstration that new users can follow to discover
  the basic functionality.
- New Setting: *Curved links*. When active, links on the Design View follow a smooth curve
  rather than a straight line.
- When execution traverses a link, a small animation is played to denote the flow of data.
  Users can set how quick they want this animation to be in Settings. The fastest setting
  effectively disables the animation.
- Special 'tag' command line arguments are now available in Tool Specification which expand
  to, for example, input database URLS or paths to optional input files when a Tool is executed.
- It is now possible to undo/redo database changes in the Data Store form.
- It is now possible to visualize the history of database changes in the Data Store form. The
  option is available in the Session menu.
- Support for Tool command line arguments. You can now give Tool (project item) command line
  arguments in addition to Tool Specification command line arguments.
- Undo/Redo in Design View
- It is now possible to add new plots to existing plot windows in Data Store View.
```

(continues on next page)

(continued from previous page)

```
- Objects in Data Store's Tree View are sorted alphabetically
- A new parameter value type, *map* has been added. There is now a dedicated editor to
  modify maps. Plotting of non-nested maps is supported, as well.
- [win-x64] importer_program.py has been built as an independent application. This␣
↪program is
  now distributed with the Spine Toolbox single-file installer. When installing Spine␣
↪Toolbox,
  the Importer Program app can be found in the Spine Toolbox install directory
  (/importer_program).
- Import preview window now supports copy-pasting mappings and options from a source␣
↪table to
  another
- Import preview window header context-menus for the preview table which, allows users to
  change all data types at once.
- Provide data for EditRole for nicer editor experience in MappingSpecModel.
- Red background is displayed for invalid values in MappingSpecModel
- Object tooltips now show the descriptions Data Store view's

### Fixed
- Data advertised by a project item during execution is only accessible by its direct
  successors. In other words, resources are passed to the next items in line but not␣
↪beyond.
- [win-x64] Executing the Importer project item has been fixed on Windows release version
- Bug fixes for Data Store View
  - Disappearing object names in entity graph
  - Spine db manager error dialogs
- Tool configuration assistant for SpineModel.jl
- [win-x64] A problem with displaying special characters in Process Log when executing␣
↪the
  Importer project item.
- The context menu in Graph view's pivot table resulted in a traceback when an entity␣
↪class
  did not have parameter definitions.
- Combobox delegate in Import preview window had wrong list of choices.
- Don't set mapping to NoneMapping if user gives a None value.
- Exporter now also exports empty object classes and empty parameters into GDX files
- A bug that sometimes made duplicate entries to File->Open recents menu
- Bug where an Excel import with empty rows would return a None in it's get_data_iterator
- [win-x64] Executing Julia tools in the embedded Julia Console
- [win-x64] Setting up Python Console. Installing ipykernel and kernel specs now works.
- [win-x64] Setting up Julia Console. Installing IJulia and kernel specs now works.
- Column indexing in Import Editor. When entering a time or time pattern index column
  manually in Import Editor's lower right corner table, the colors in the preview table
  failed to update. This should fix the bug and allow column selection both by column␣
↪index
  or column header text.

### Changed
- spinetoolbox is now a Python package. To start the app, use command
  `python -m spinetoolbox` or `python spinetoolbox.py` as spinetoolbox.py has been moved␣
↪to
  repository root.
```

(continues on next page)

(continued from previous page)

```
- Tool templates are now called Tool specifications
- File->Open Project opens a file dialog, where you can open projects by selecting an old
  <project_name>.proj file or a Spine Toolbox Project directory. Valid Spine Toolbox␣
↪projects
  are decorated with the Spine logo.
- Old style projects (.proj files) cannot be opened anymore. The old style projects need␣
↪to
  be upgraded before opening. You can upgrade your .proj file projects into new ones with
  *Project Upgrade Wizard* found in `File->Upgrade project` menu item.
- Project information is not saved to a <project_name>.proj file anymore. This␣
↪information
  is now located in file <project_dir>/.spinetoolbox/project.json. Every Spine Toolbox␣
↪project
  has this file.
- Work directory is now a global setting instead of a project setting
- Renamed *Data Interface* project item to *Importer*. The corresponding category
  *Data Importers* was renamed to *Importers*.
- Tree, graph, and tabular views have been merged into one consolidated view. You can␣
↪choose
  your preferred style from the Data Store View's `View` menu.
- The graph view behavior has changed. Now selecting objects in the object tree not only
  shows those objects but also all the cascading relationships. This is to facilitate␣
↪exploring
  the system without a previous knowledge.
- importer_program.py now uses the Python interpreter that the app was started with and␣
↪not
  the one that is given by user in Settings -> Tools.
- Importer now uses QProcessExecutionManager for running the importer_program.py

### Removed
- The status bar of the Data store view is gone. Instead, notifications are printed in a␣
↪box
  on the right side of the form.
- Saving project information to .proj files is not happening anymore

## [0.3] - 2019-09-06

### Added
- Welcome message including a link to Getting Started guide.
- Zooming (by using the mouse wheel) is now enabled in Design View. You can also select␣
↪multiple project.
  items by pressing the Ctrl-key down and dragging the mouse.
- New project item icons on Design View.
- Two options for the Design View background (grid or solid). See Settings (F1).
- Python Console (menu View -> Dock Widgets -> Python Console).
- Python interpreter can be selected in Settings (Also Python 2.7 supported).
- Support for Python Tool templates
- Python Tool execution using the command line approach.
- Python Tool execution using the embedded Python Console (checkbox in Settings)
- The Data Store treeview now also has a relationship tree.
- Support for reordering columns in the treeview.
- Selecting 'edit object classes' in the treeview now also allows the icon to be␣
```

(continues on next page)

(continued from previous page)

```
↪selected.
- Play button to main window Toolbar that executes all Directed Acyclic Graphs in the␣
↪Project one after another.
- Another Play button to main window Toolbar to execute selected Directed Acyclic Graph.␣
↪Selecting a DAG happens by
  selecting one or more project items belonging to the wanted DAG in the Design View or␣
↪in Project Items list.
- Stop button to main window Toolbar, which terminates execution.
- Possibility to specify a dedicated Julia project or environment for Spine Toolbox in␣
↪the settings.
- Feature to Export project to GraphML format. Each graph in Design View is written to␣
↪its own file. To do this, just
  select *Export project to GraphML* from the Project Item list context-menu or from␣
↪*File ->
  Export project to GraphML*.
- New project item: *Data Interface*
- Parameter and relationship parameter values can now be edited in a dedicated editor␣
↪window in Tree, Graph and Tabular
  views. The editor is accessible by right-clicking a value and selecting `Open in␣
↪editor...`.
- It is now possible to plot parameter and relationship parameter values in Tree, Graph␣
↪and Tabular views.

### Fixed
- There is now an upper limit on how much text is logged in Event Log and Process Log.␣
↪The oldest lines are removed
  if the limit is exceeded. This fixes a problem with excessive memory usage when␣
↪running long processes.
- Mouse click problems in Design View
- Mouse cursor problem in Tool Configuration Assistant
- Welcome message is now shown for first time users
- NameError: SpineDBAPIError when executing Data Stores.
- .py files in *spinedb_api\alembic\versions* are now copied to the installation bundle␣
↪without compiling them to
  .pyc files first. Also, if there's a previous version installed, *spinedb_api\alembic*␣
↪directory is deleted
  before installation begins [Win-x64].
- Added missing modules from *spinedb_api\alembic\versions* package into installation␣
↪bundle [Win-x64].
- *numpy* is now deleted before installation begins so installing over an existing␣
↪installation of Spine Toolbox
  works again [Win-x64].
- *packaging* and *appdirs* packages are now included in the installation bundle [Win-
↪x64].

### Changed
- Selecting the Julia environment in Settings now requires picking the Julia interpreter␣
↪**file**
  (e.g. julia.exe on Windows) instead of the directory where the Julia interpreter is␣
↪located.
- Selecting the GAMS program (**file**) in Settings now requires picking the GAMS␣
↪program (e.g. gams.exe
```

(continues on next page)

(continued from previous page)

```
  on Windows) instead of the directory where the GAMS program is located.
- All application Settings are now saved using Qt's QSettings class. Old configuration␣
↪file,
  *conf/settings.conf* file has been removed.
- New Spine databases can be created in any backend supported by spinedb_api. The Data␣
↪Store item properties
  have been changed to allow for this.
- Executing Directed Acyclic Graphs instead of just Tools.
- Executing project items does not happen from the Tool Properties anymore. It happens␣
↪instead by pressing the
  Play button in the main window Toolbar.

### Removed
- An unnecessary error dialog in Import Preview widget.
- ConfigurationParser class in configuration.py.
- Execute button in Tool Properties.
- Stop button in Tool Properties.
- Console window that opened in addition to the application window is not shown anymore␣
↪[Win-x64].

## [0.2] - 2019-01-17

### Added
- New Setting. You can now select whether to delete project item's data directory
  when removing a project item.
- Graph View is also available from Data Store items, allowing to insert new
  relationships more graphically.
- You can now execute Tools in the work directory or the source directory (where the
  main program file is located). The Setting is in the Tool template editor and in Tool
  properties.
- Tabular view: New view for Data Store items, allowing to view and edit data in a
  pivot table.
- Tool Properties now shows all information about the attached Tool template
- Context-menus for Data Connection Properties, Tool Properties, and View Properties
- Support for optional input files for Tool templates. You can now use Unix style␣
↪wildcards (`*` and `?`)
  to specify the optional files that a Tool may exploit, e.g. `*.csv`.
- Wildcard support for Tool template output files
- Tool template output files now support subdirectories
- You can now create a new (blank) main program file in the Tool template editor by␣
↪pressing the the *+* button
  and selecting `Make new main program`
- A shortcut to Tool template main directory, accessible e.g. in Tool Properties context-
↪menu
- New Setting. You can select whether the zoom in the Graph view is smooth or discrete
- Windows installer default location is /Program Files/SpineToolbox/. When new versions␣
↪are released, the
  installer will automatically upgrade Spine Toolbox to a new version in that directory.
- Support for executing Executable type tools in Linux&Mac. (Windows support was added␣
↪previously)
- Tool configuration assistant. Available in menu `File->Tool configuration assistant`.␣
↪Checks automatically
```

(continues on next page)

(continued from previous page)

```
  if the Julia you have selected in Settings supports running Spine Model. If not, the␣
↪required packages are
  automatically installed.

### Fixed
- Better support for scaling screen resolutions
- Exporting a datapackage to Spine format failed if datapackage.json was not explicitly␣
↪saved

### Changed
- Importing items with names into a spinedatabase moved to spinedatabase_api to
  allow for easier adding of new import formats in future versions.
- Tool template list is now located in the Project dock widget below the list of Project␣
↪items
- New look for Spine Datapackage editor

### Removed
- Templates tab in `Project` dock widget.

## [0.1.75] - 2018-11-23

### Added
- New Setting (F1). You can now select whether to delete project item's data
  directory when removing a project item.
- Application icon (Spine symbol)
- New installer for 64-bit Windows:
    - Installation file extension is now *.exe* instead of *.msi*
    - Show license file before installation (users must agree to continue)
    - Default install folder is now `C:\Program Files\ `.
    - **No** need to *Run as Administrator* even if installed to the default location
    because write permissions for sub-folders that need them (\conf, \projects,
    \work) are set automatically
    - Create a shortcut on desktop (if wanted)
    - Create a Start Menu folder (if wanted)
    - Uninstaller. Available in the Start Menu folder or in Windows
    Add/Remove Programs
    - Remove app related registry entries when uninstalling (if wanted)

### Fixed
- Data Package editor. Some files were missing from the tabulator package.
- Bug when exiting the app when there is no project open in close_view_forms() when
  exiting the application without a project open

### Changed
- settings.conf file is now in /conf directory

## [0.1.71] - 2018-11-19

### Added
- Added PyMySQL package, which was missing from the previous release
- Improved Graph View for the View project item (work in progress)
```

(continues on next page)

(continued from previous page)

```
### Changed
- Some main window components have been renamed
    - Main view is now called *Design View*
    - Julia REPL is now called *Julia Console*
    - Subprocess Output is now called *Process Log*
    - Project item info window is now called *Properties*

## [0.1.7] - 2018-11-01

### Added
- New option to refresh the data store tree view and get latest changes from the␣
↪database.
- Several performance enhancements in tree view form (accessing internal data more␣
↪efficiently,
  optimizing queries to the database.)
- Now the data store tree view offers to commit pending changes at exit.
- Better support for batch operations in data store tree view.
- data store tree view can be fully operated by using the keyboard.
- New options to edit object tree items in the data store tree view, including changing␣
↪the objects involved
  in a relationship.
- The dialogs to add/edit tree view items are not closed in case of an error, so the␣
↪user can adjust their choices
  and try again.
- Stop button now terminates tool execution.
- New context menu options to fully expand and collapse object tree items in the data␣
↪store tree view.
- The autofilter in the data store tree view now also filters work in progress rows.
- In the Data store item controls, the path to the SQLite file can be specified by␣
↪dropping a file.
- Parameter and parameter value tables in the data store tree view now have an empty row␣
↪at the end,
  which can be used to enter data more easily.
- JSON data can be visualized and edited more easily in the data store tree view.
- Tools can now execute (Windows) batch files and other executables (.exe). Linux␣
↪support pending.
- About Qt dialog added to Help menu

### Fixed
- Clicking on the open treeview button while the data store tree view is open now raises␣
↪it, rather than opening a
  second one.
- Work folder is not created for Tools if the Tool template requirements are not met.
- Result folder is not created if the Tool template fails to start.
- The embedded Julia REPL now uses the Julia that is given in application Settings (F1).
  Previously, this used the default Julia of your OS.

### Removed
- Connections tab has been removed (actually, it is just hidden and can be restored with␣
↪a keyboard shortcut)
- Refresh Tools button on Templates tab has been removed as it was not needed anymore
- Set Debug message level in Settings has been removed
```

(continues on next page)

(continued from previous page)

```
## [0.1.5] - 2018-09-28

### Added
- Advanced copy/pasting and multiple selection support for the data store tree view.
- Import data from Excel files into the data store tree view.
- Export Spine database from the data store tree view into an Excel file.
- Save at exit prompt.
- Import data from datapackage into the data store tree view.
- Restore Dock Widgets in the main window.
- Parameter tables can be filtered by clicking on their headings in the data store tree␣
↪view.
- Parameter and parameter values are added and edited directly in the data store tree␣
↪view,
  without need for an additional dialog.
- On-the-fly creation of necessary relationships when entering parameters in data store␣
↪tree view.
- View item feature for visualizing networks from a Spine database. A view item can␣
↪visualize databases
  from all data store items connected to it.
- Packages numpy, scipy, and matplotlib are now mandatory requirements.
- Drag files between data connections. File items can be dragged from the references and␣
↪data lists.
  Data connection items can be selected by hovering them while dragging a file. Dropping␣
↪files onto a Data Connection
  item copies them to its data directory.
- datapackage.json files in data connections are now opened with the Spine datapackage␣
↪form. This is a dedicated
  interface to prepare the datapackage for importing in the data store tree view.
- The data store tree view does not lock the database when there are uncommitted changes␣
↪anymore.

### Changed
- Changed DBAPI package mysqlclient (GPL license, not good) to pymysql (MIT license,␣
↪good)
- spinedatabase_api is not included in Spine Toolbox repository anymore. It is a required
  package from now on.
- Data Store item can have only one database, not many. When opening a project created␣
↪with a
  previous version, the first database in the list of saved references will be loaded␣
↪for each Data Store.
- In the data store tree view, the object tree presents all relationship classes at the␣
↪same level,
  regardless of how many object classes are involved. The same applies for relationships␣
↪and objects.
- In the data store tree view, the relationship parameter value view now has different␣
↪columns for each object
  in the relationship.

## [0.1] - 2018-08-20

### Added
```

(continues on next page)

(continued from previous page)

```
- Basic functionality
```

CHAPTER

TWO

GETTING STARTED

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

This chapter introduces the following topics:

2.1 Spine Toolbox Interface

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

- **Properties** provides an interface to interact with the currently selected project item.
- **Event Log** shows relevant messages about user performed actions and the status of executions.
- **Console** allows Spine Toolbox to execute Tools written in Python, Julia or GAMS and provides an interface to interact with the aforementioned programming languages. Also shows a list of parallel executions available in the project.

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

Tip: You can drag-and-drop the dock widgets around the screen, customizing the interface at your will. Also, you can select which ones are shown/hidden using either the **View -> Dock Widgets** menu, or the main menu **Toolbar**'s

context menu. Spine Toolbox remembers your configuration between sessions. Selecting **Restore Dock Widgets** from the **View -> Dock Widgets** menu restores the widgets back to their default location.

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

2.2 Creating a Project

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

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

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

Congratulations, you have created your first Spine Toolbox project.

2.3 Creating a Tool Specification

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

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

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

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

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

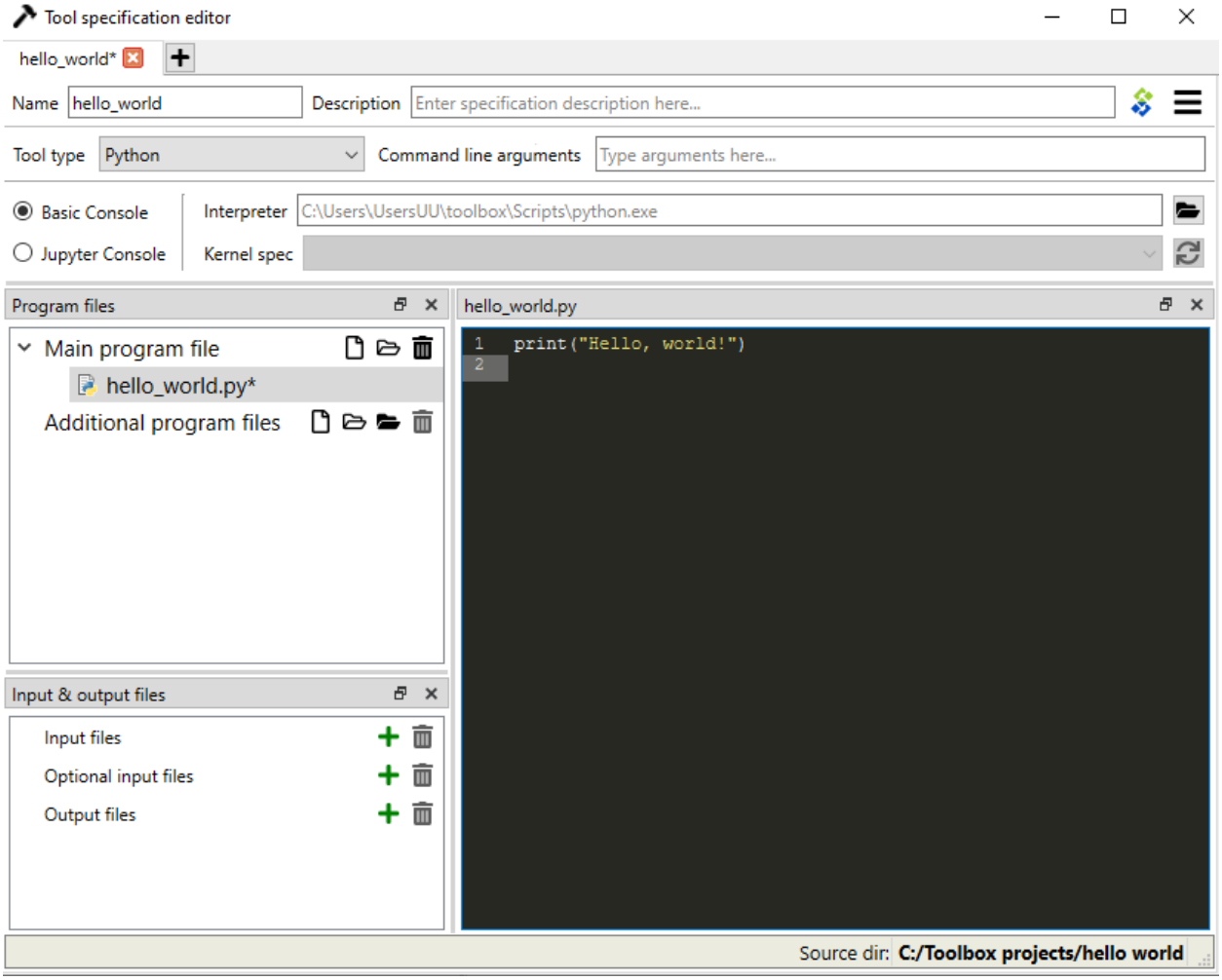
Select `hello_world.py` below the *Main Program File*. Click on the (black) editor dock widget with the title 'hello_world.py'.

Type in the following:

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

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

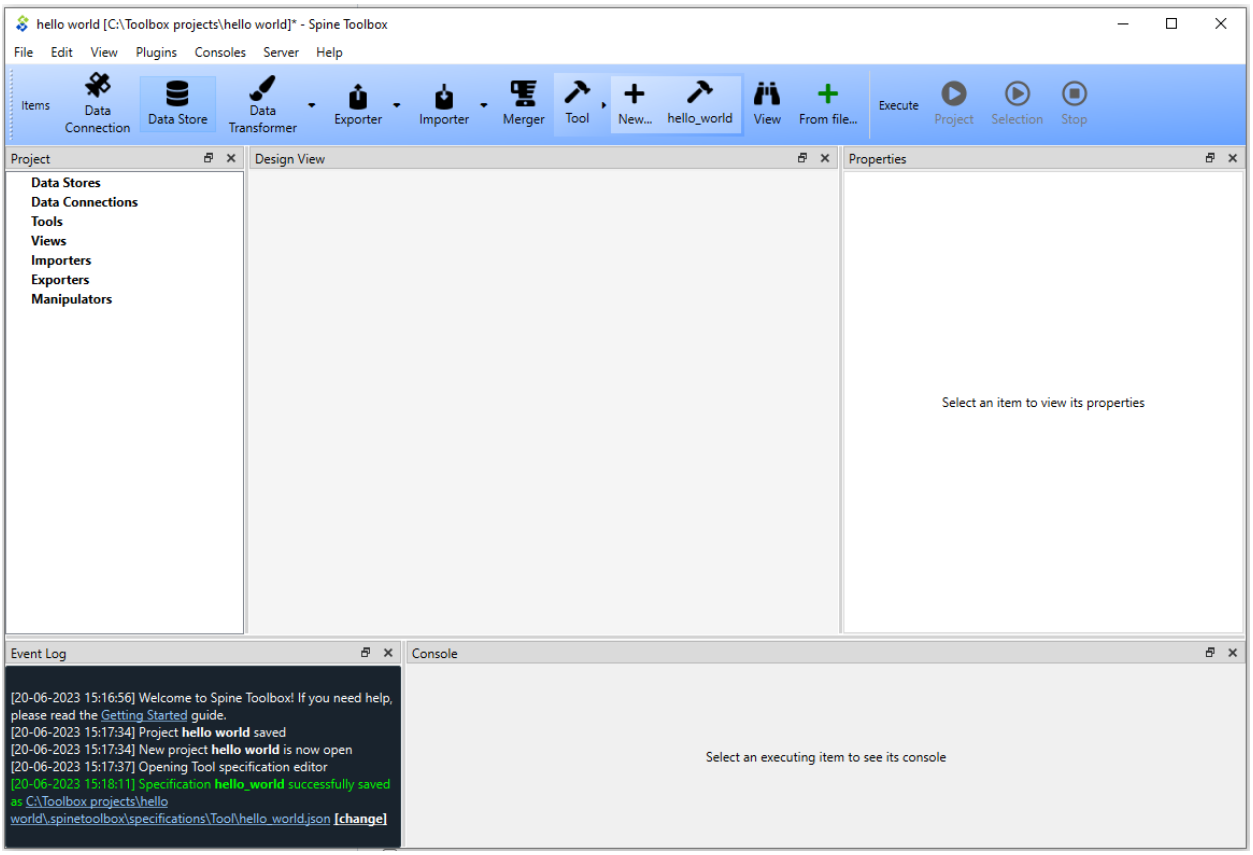
The **Tool specification editor** should be looking similar to this now:



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

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

Your main window should look similar to this now.



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

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

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

Congratulations, you have just created your first Tool specification.

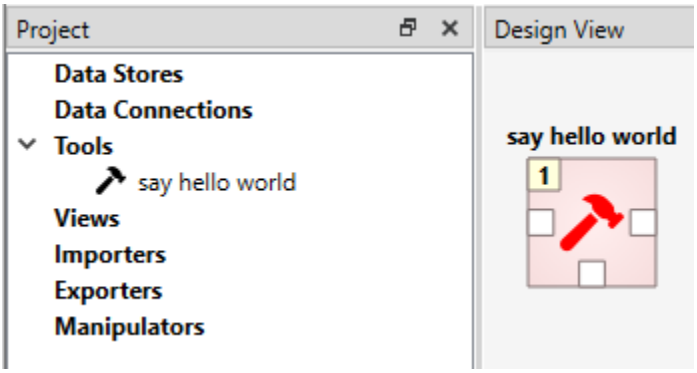
2.4 Adding a Tool Item to the Project

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

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

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

It should look similar to this:

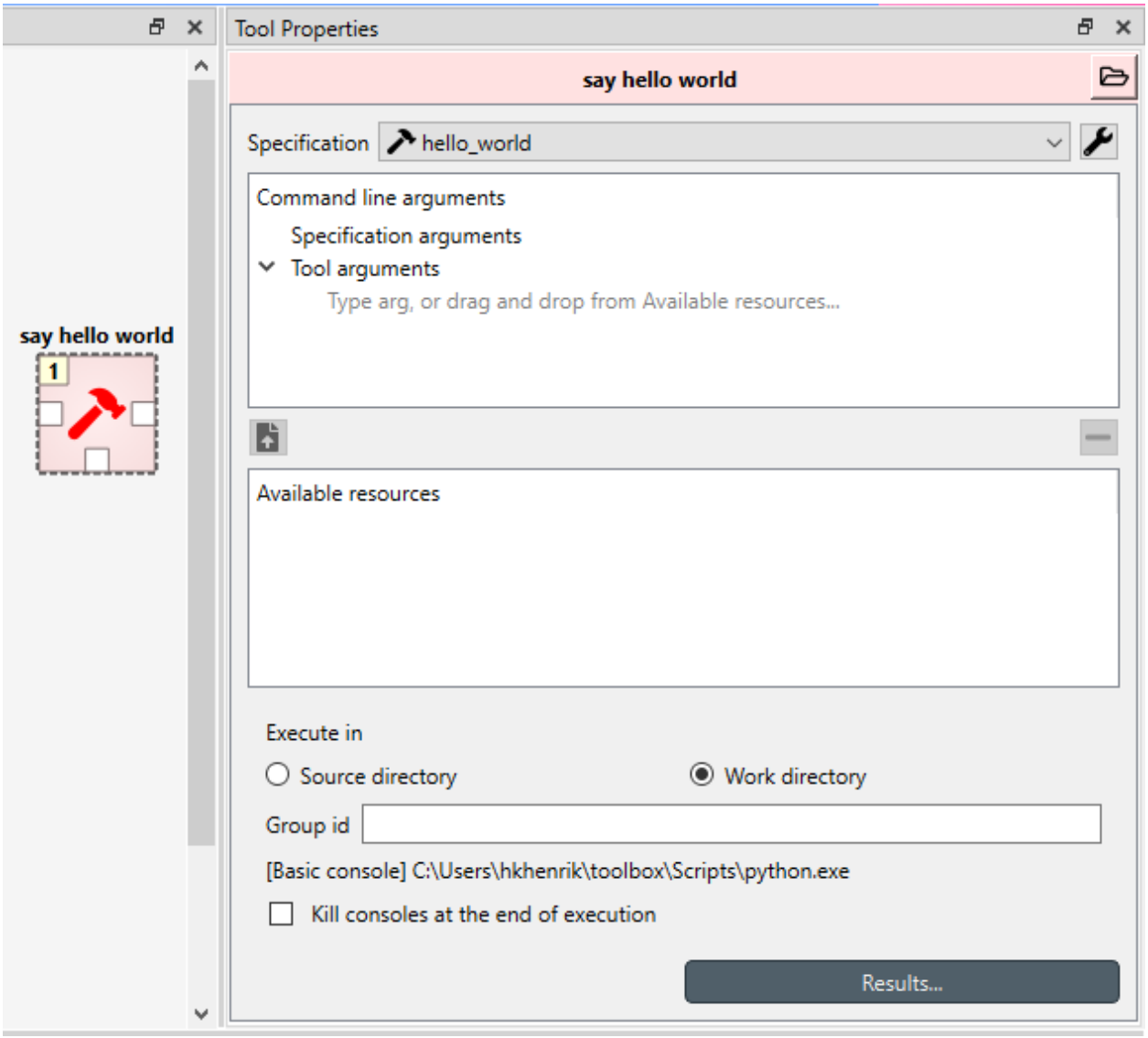


Another way to do the same thing is to drag the with the 'hello world' text from the **Toolbar** onto the **Design View**. Similarly, the **Add Tool** form will popup but the 'hello world' tool specification is already selected from the dropdown list.

Note: The Tool specification is now saved to disk but the project itself is not. Remember to save the project every once in a while when you are working. You can do this by selecting **File -> Save project** from the main window or by pressing **Ctrl+S** when the main window is active. If the project is in such a state that it has unsaved changes, an asterisk * is visible after the project name and path in the upper left corner of the main window.

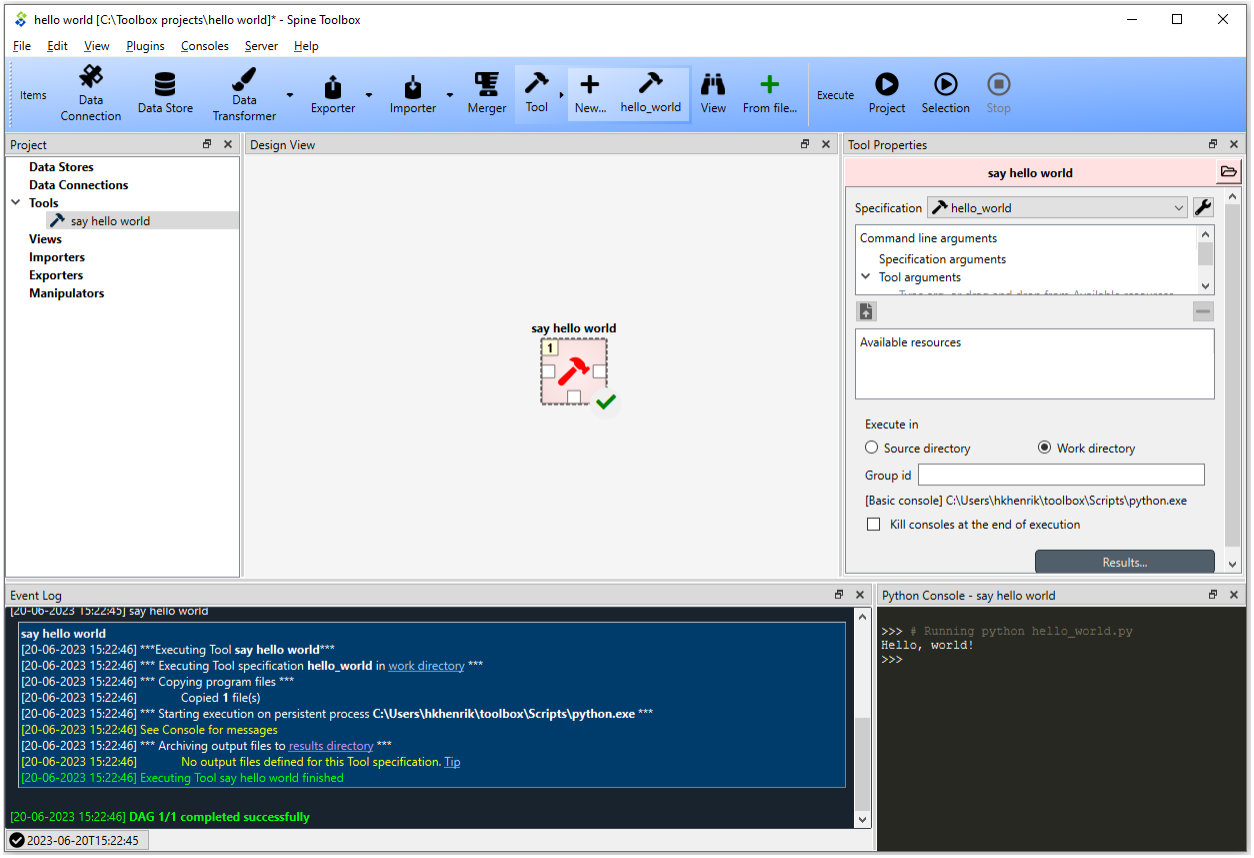
2.5 Executing a Tool

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



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

Once the execution is finished, you can see the details about the item execution as well as the whole execution in **Event Log**. The **Console** contains the output of the executed program file.



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

Congratulations, you just executed your first Spine Toolbox project.

2.6 Editing a Tool Specification

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

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

Open the Tool specification editor for the 'hello world' Tool spec. You can do this for example, by double-clicking the 'say hello world' Tool in **Design View**, or from the **Tool Properties** by clicking the Tool specification options button () next to the specification and selecting **Edit specification**.

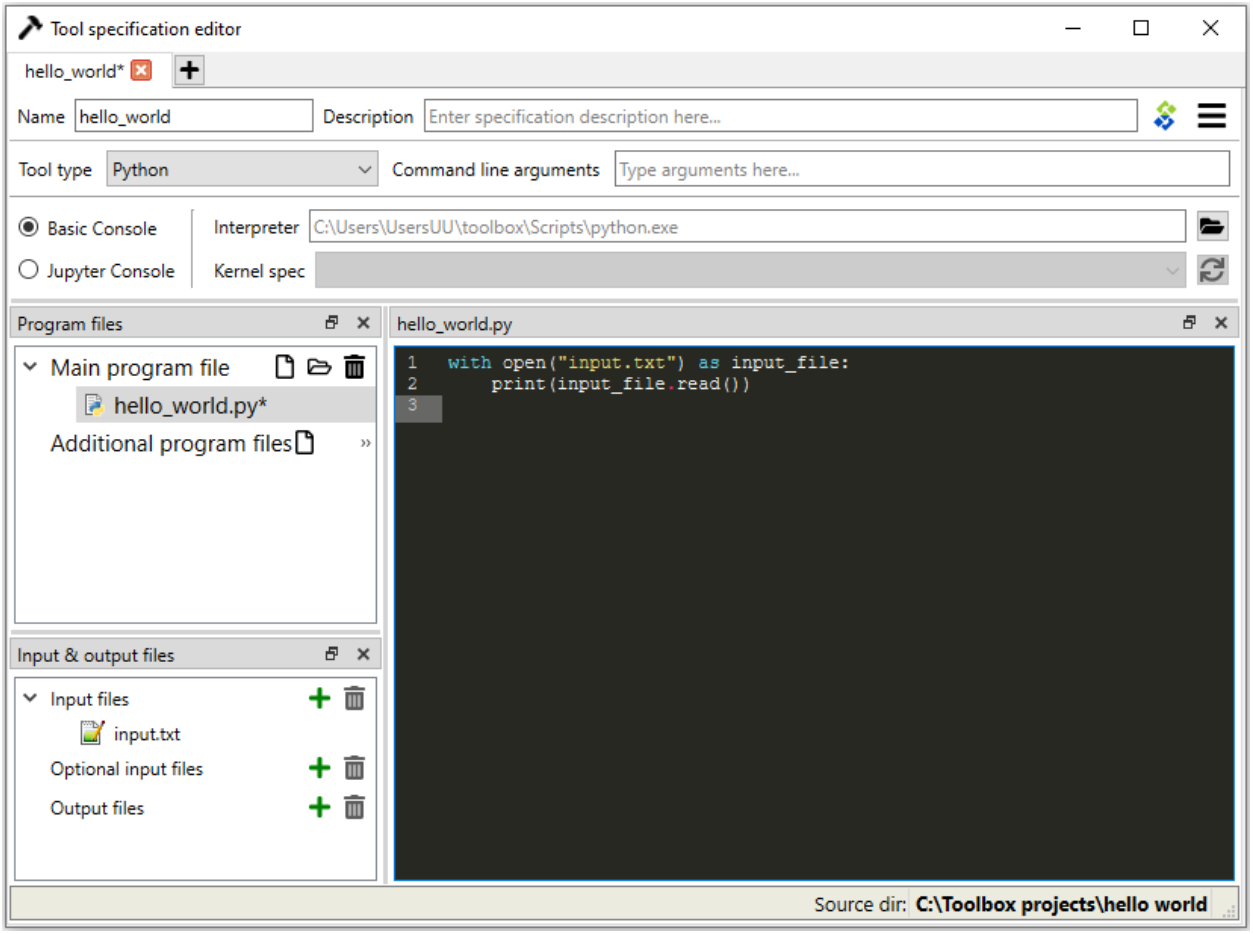
In **Input & Output files** dock widget, click the button next to the *Input Files* text. A dialog appears, that lets you enter a name for an input file. Type 'input.txt' and press Enter.

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

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

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

The editor should now look like this:

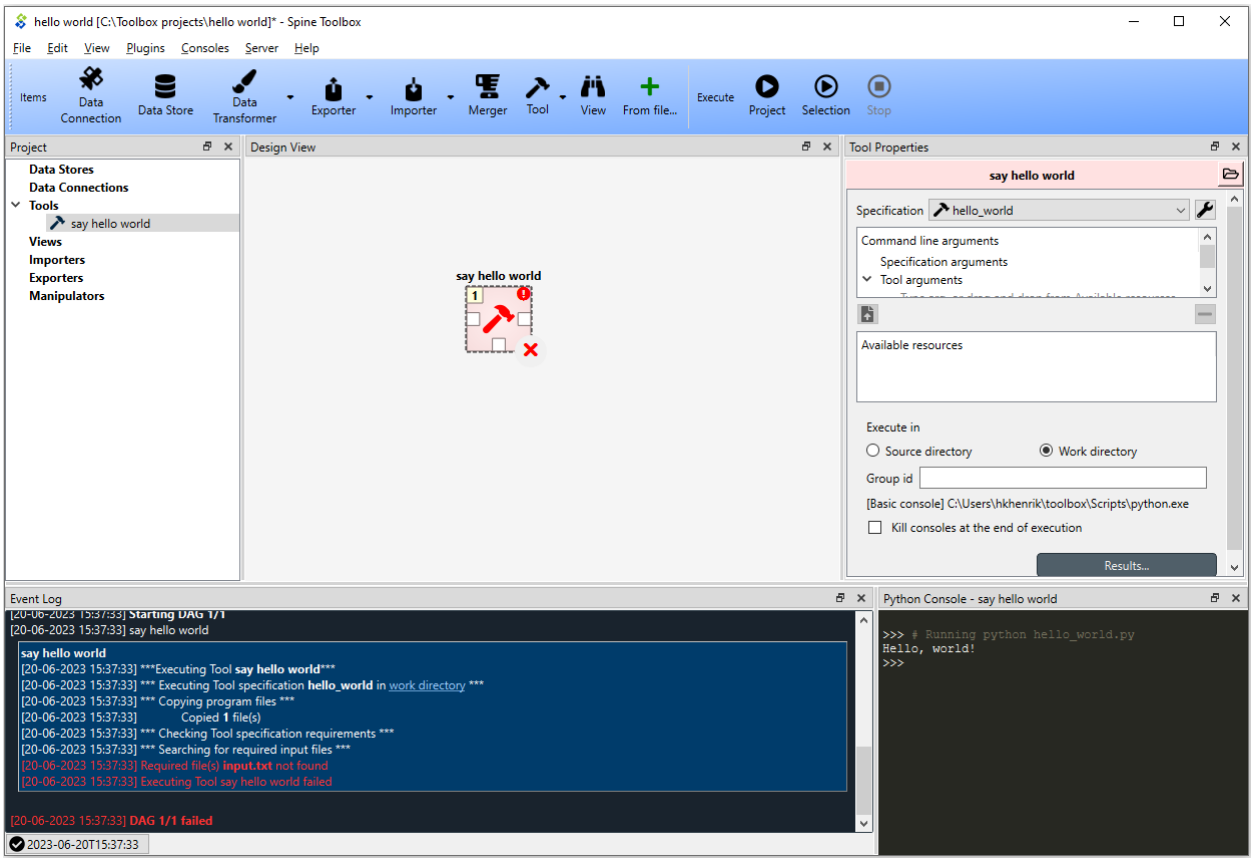


Save the specification and close the editor by pressing **Ctrl+S** and then **Alt+F4**.

Note: See *Tool Specification Editor* for more information on editing Tool specifications.

Back in the main window, note the exclamation mark on the Tool icon in **Design View**, if you hover the mouse over this mark, you will see a tooltip telling you in detail what is wrong. If you want you can try and execute the Tool anyway

by pressing in the **Toolbar**. *The execution will fail* because the file `input.txt` is not made available for the Tool:



2.7 Adding a Data Connection Item to the Project

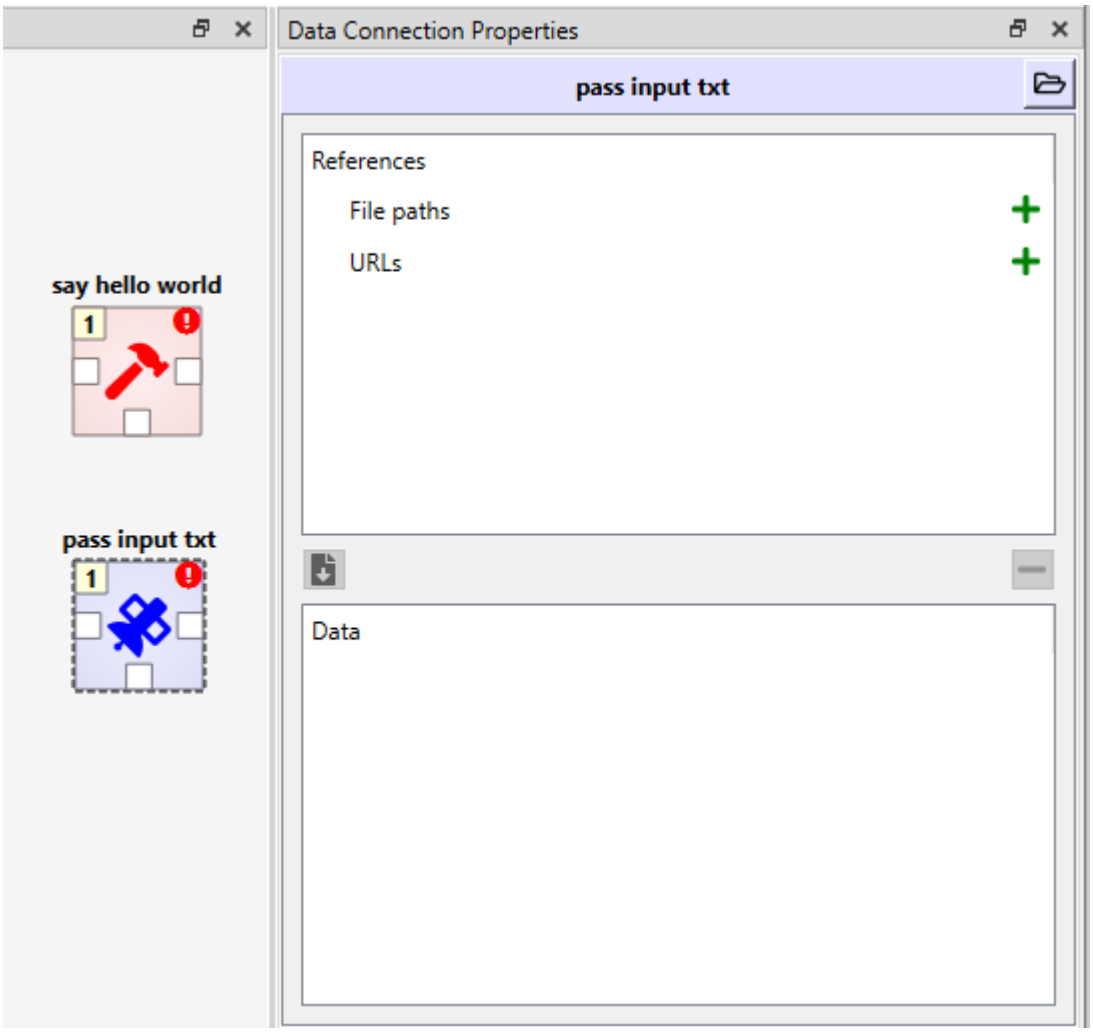
Note: The Data Connection item is used to hold generic data files, so that other items, notably Importer and Tool, can make use of that data.

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

The *Add Data Connection* form will show up. Type 'pass input txt' in the name field and click **Ok**. The newly added Data Connection item is now in the **Design View**.

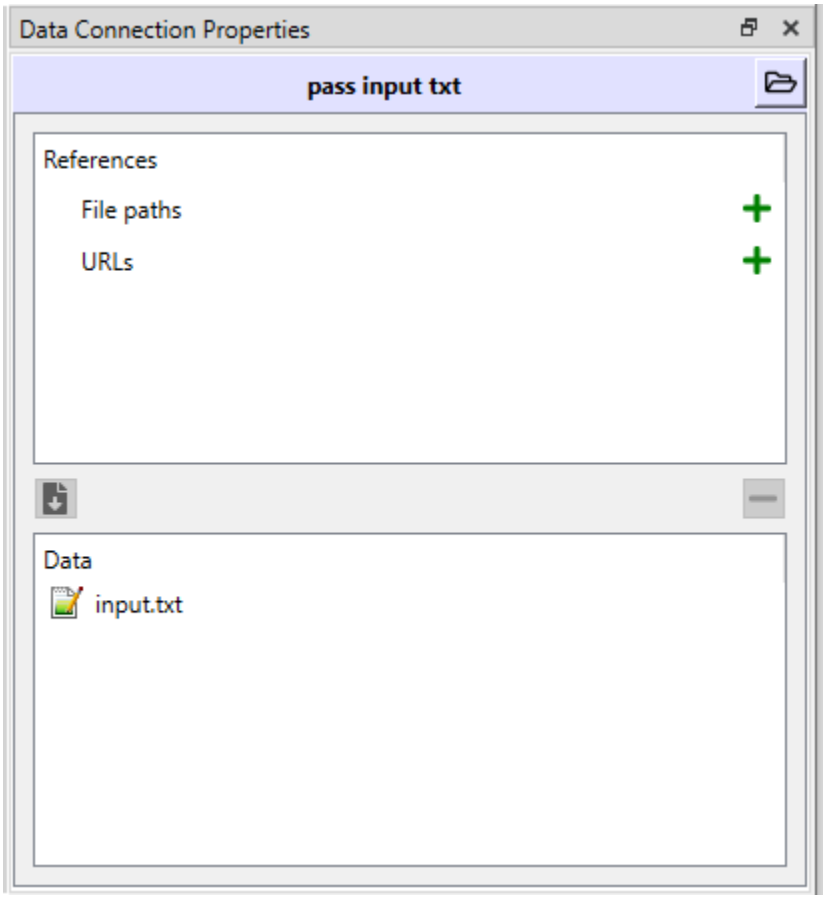
2.8 Adding Data Files to a Data Connection

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

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



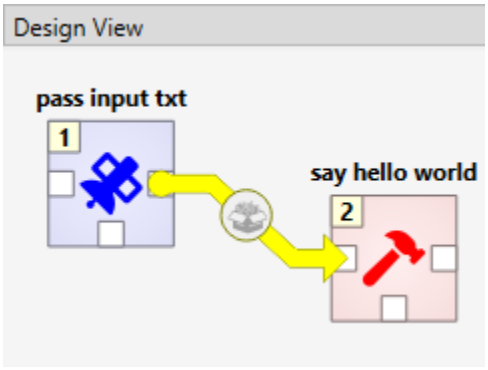
Double click this file to open it in your default text editor. Then enter the following into the file's content:

```
Hello again, World!
```

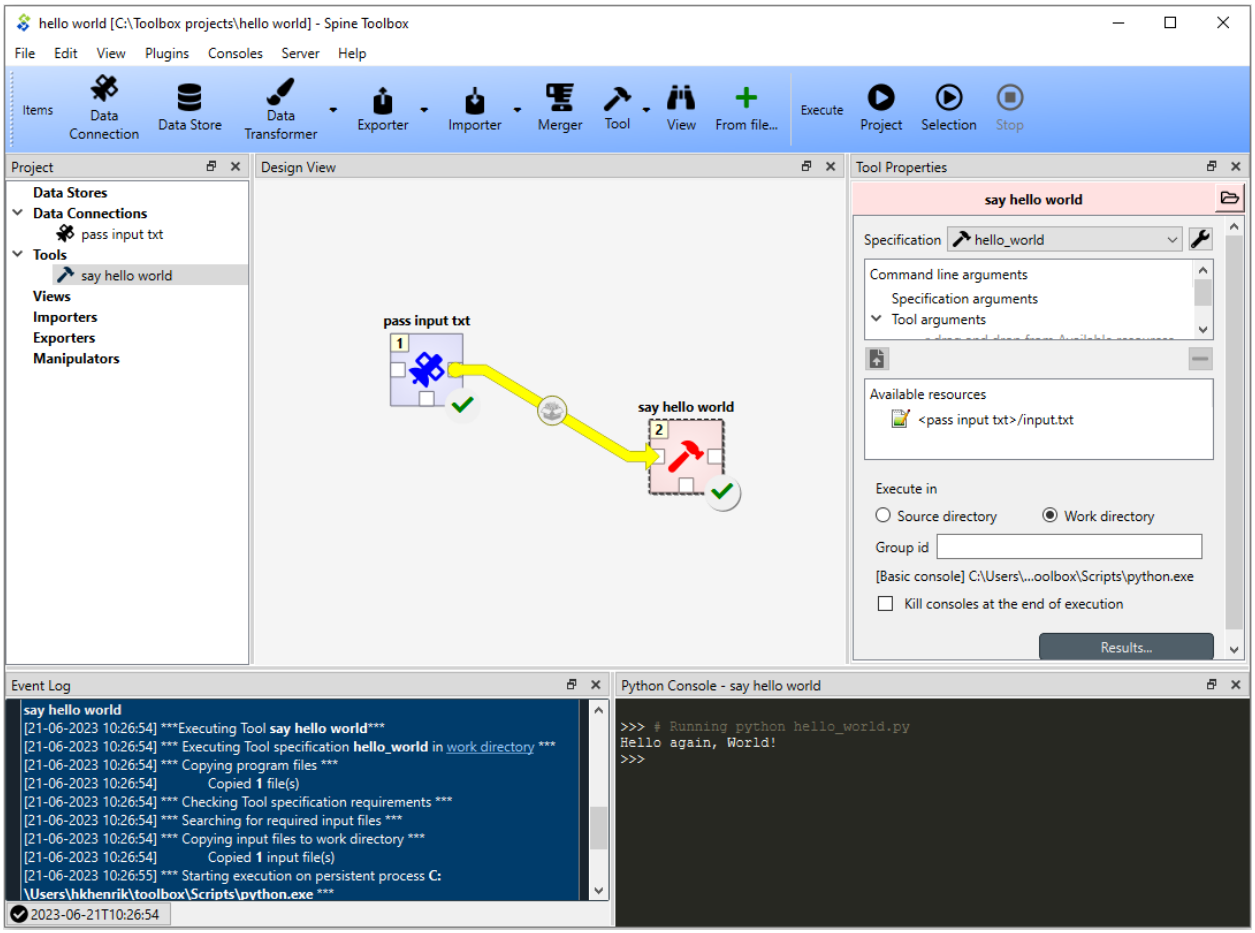
Save the file.

2.9 Connecting Project Items

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



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



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

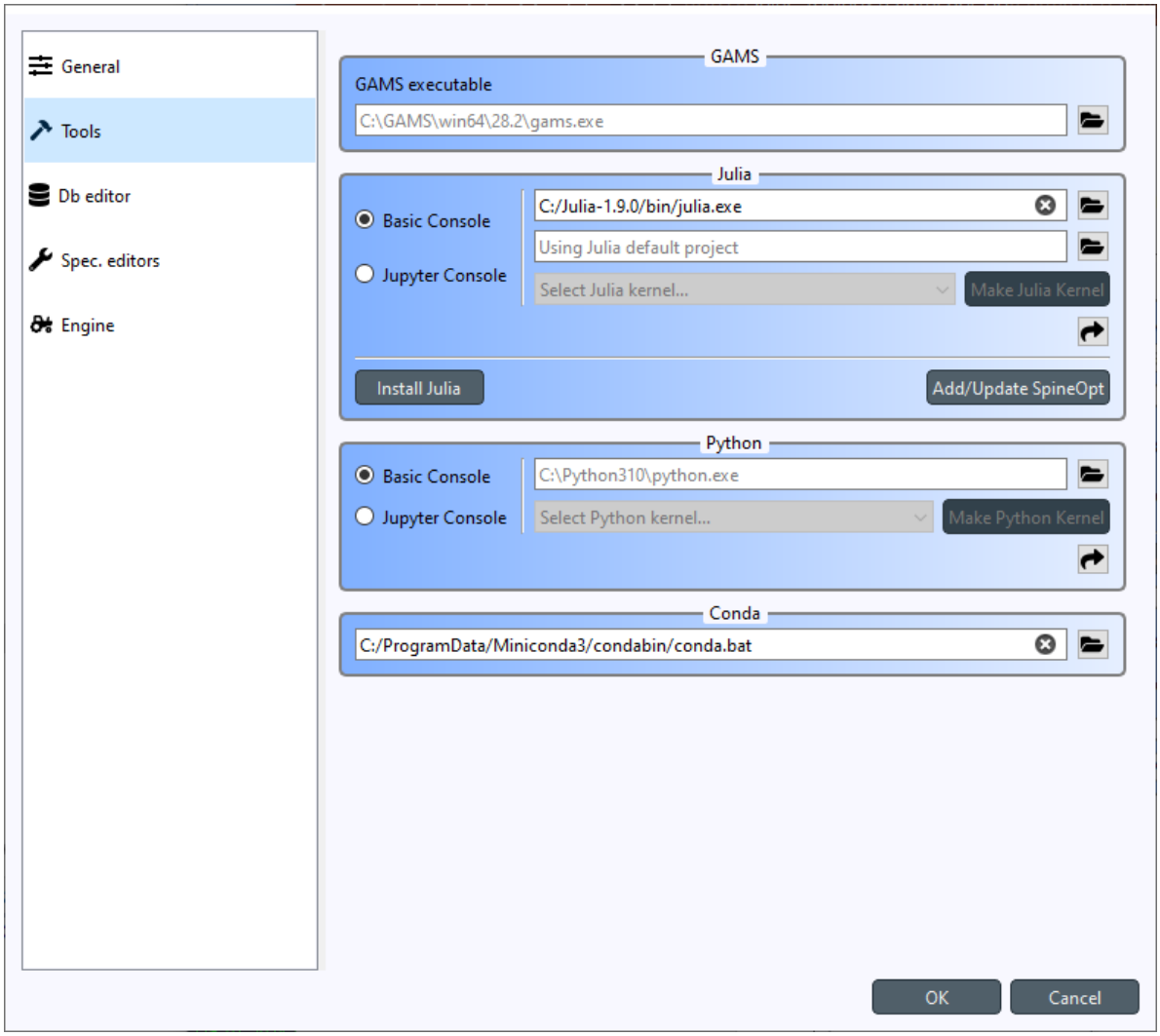
Where to next: If you need help on how to set up and run **SpineOpt.jl** using Spine Toolbox, see chapter *How to Set up SpineOpt.jl*. After setting up SpineOpt, there are three tutorials over on **SpineOpt.jl**'s documentation that will help you get started on using SpineOpt in Spine Toolbox: Simple system, Two hydro plants, and Case study A5.

CHAPTER

THREE

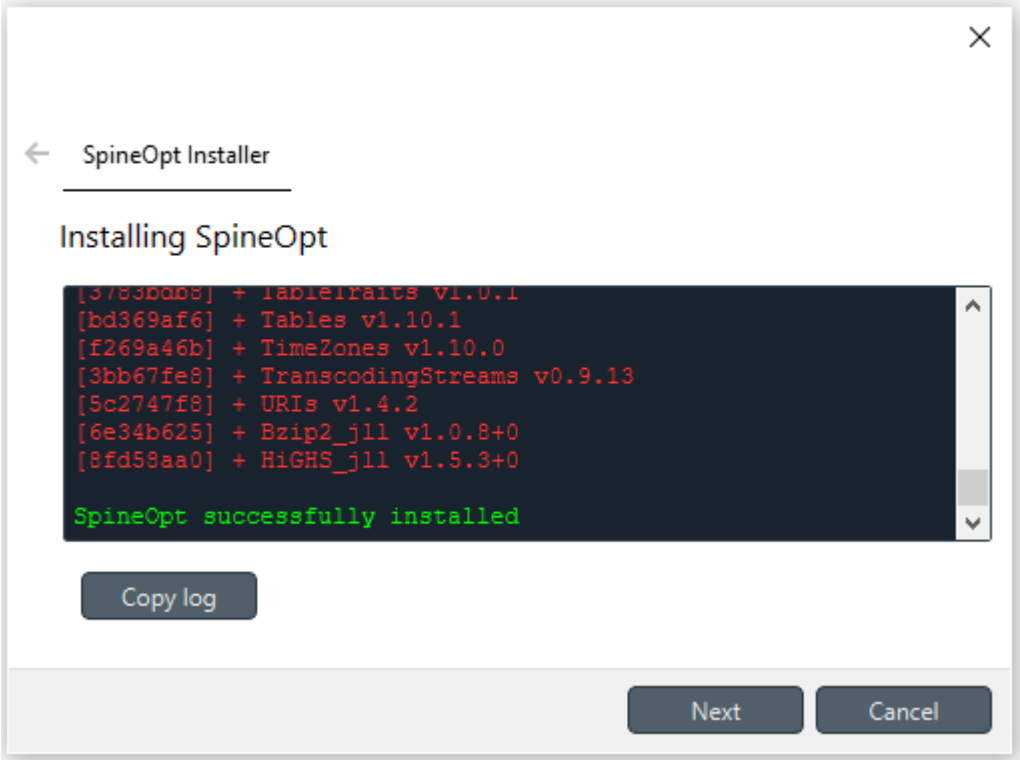
HOW TO SET UP SPINEOPT.JL

1. Install Julia (v1.6 or later) from https://julialang.org/downloads/ if you don't have one. See latest **SpineOpt.jl** Julia compatibility information here.
2. Start Spine Toolbox
3. Create a new project (**File -> New project...**)
4. Select **File -> Settings** from the main menu and open the *Tools* page.
5. Set a path to a Julia executable to the appropriate line edit (e.g. *C:/Julia-1.6.0/bin/julia.exe*). Your selections should look similar to this now.



6. *[Optional]* If you want to install and run SpineOpt in a specific Julia project environment (the place for *Project.toml* and *Manifest.toml*), you can set the path to the environment folder to the line edit just below the Julia executable (the one that says *Using Julia default project*).

7. Next, you need to install **SpineOpt.jl** package for the Julia you just selected for Spine Toolbox. You can do this manually by following the instructions **or** you can install **SpineOpt.jl** by clicking the **Add/Update SpineOpt** button. After clicking the button, an install/upgrade SpineOpt wizard appears. Click **Next** twice and finally **Install SpineOpt**. **Wait until the process has finished** and you are greeted with this screen.



Close the wizard.

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

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

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

CHAPTER

FOUR

SETTING UP CONSOLES AND EXTERNAL TOOLS

This section describes the options for executing different Python, Julia, Gams, and Executable Tools and how to set them up. To get started with **SpineOpt.jl**, see *How to Set up SpineOpt.jl*. See also *Executing Projects*.

Python and Julia Tools can be executed either in an embedded *Basic Console* or in a *Jupyter Console*. GAMS Tools are executed in a sub-process. Executable Tools (external programs) are executed in a shell or by running the executable file directly. You can also make a Tool that executes a shell command by creating an *Executable* Tool Spec in **Tool Specification Editor**, entering the shell command to the *Command* line edit and then selecting the Shell for this Tool.

4.1 Basic Consoles

Basic Console appears to the **Console** dock widget in the main window when you execute () a project containing either a Python or a Julia Tool with Basic Console selected.

4.1.1 Python

Executing a Python Tool in the Basic Console requires no set up. Simply create a *Python* Tool Spec in **Tool Specification Editor** and select the Basic Console radio button. The default Python interpreter used in launching the Console is the same Python that was used in launching Spine Toolbox. You can also select another Python by changing the Python interpreter line edit. Remember to save the new Tool Spec when closing the **Tool Spec. Editor**. Then drag the Python Tool Spec into the **Design View**, and press to execute it.

Note: The Python settings on the *Tools* page in **File -> Settings** are the *default* settings for new Python Tool Specs. You can select a different Python executable for each Python Tool Spec separately using the **Tool Specification Editor**.

4.1.2 Julia

To execute Julia Tools in the Basic Console, first install Julia (v1.6 or later recommended) from here and add *<julia install path>/bin* to your PATH environment variable (if not done automatically by the installer). Then go to the *Tools* page in **File -> Settings** and make sure that the Basic Console radio button is selected in the Julia group. If Julia is in your PATH, the Julia executable line edit should show the path as (grey) placeholder text. If you want to use another Julia on your system, you can change the path in the line edit. You can also set a Julia Project below the Julia executable line edit.

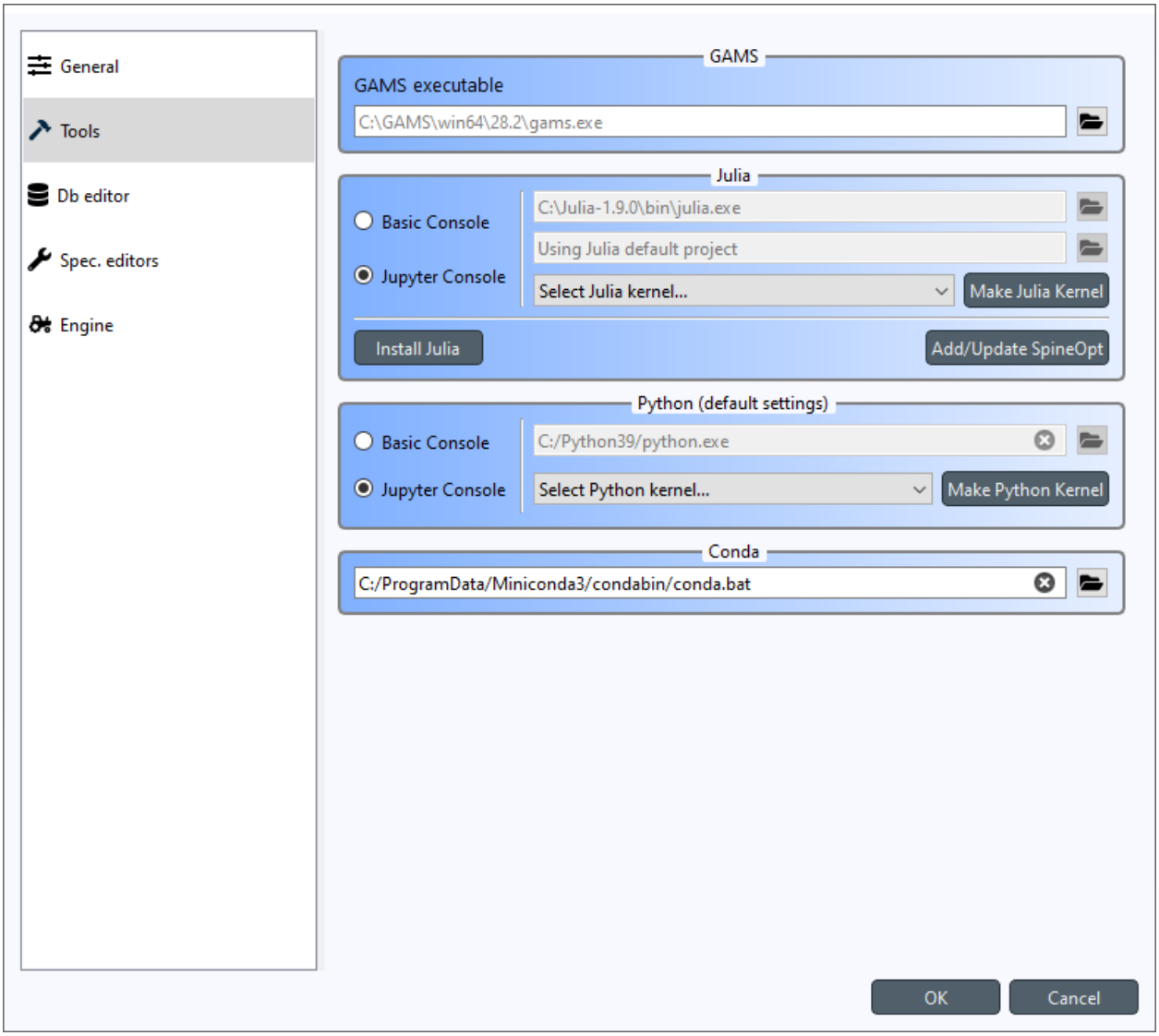
Note: The Julia settings on the *Tools* page in **File -> Settings** are the *default* settings for new Julia Tool Specs. You can select a different Julia executable & project for each Julia Tool Spec separately using the **Tool Specification Editor**.

4.2 Jupyter Consoles

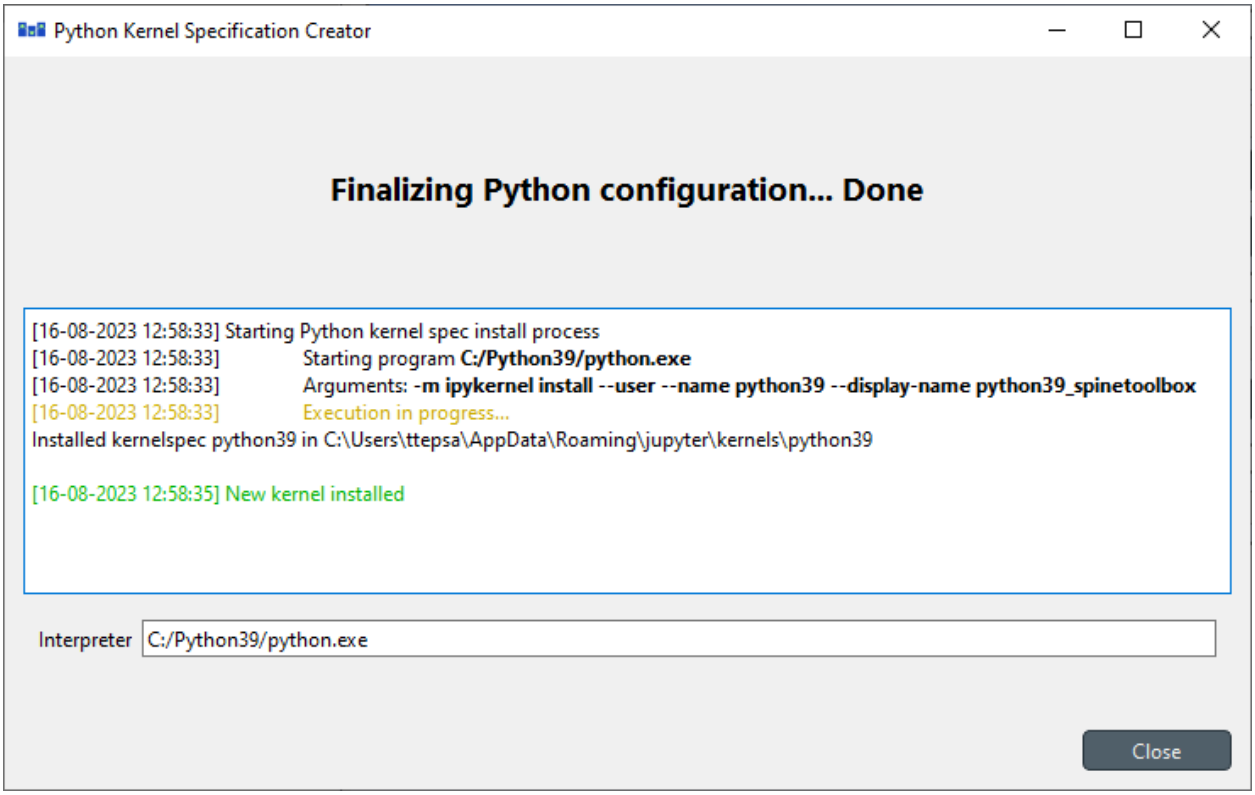
Jupyter Console appears to the **Console** dock widget in the main window when you execute () a project containing either a Python or a Julia Tool with the *Jupyter Console* selected. The Jupyter Console requires a Jupyter kernel to be installed on your system. Kernels are programming language specific processes that run independently and interact with the Jupyter Applications and their user interfaces.

4.2.1 Python

Select *Jupyter Console* radio button in **File -> Settings**. You also need to select the Python kernel you wish to use from the *Select Python kernel...* combo box. If this list is empty, you need to install the kernel specs on your system. You can either do this manually or click the **Make Python Kernel** button. Clicking the button opens a **Python Kernel Specification Creator** window, that first installs the **ipykernel** package (if missing) for the Python that is currently selected in the Python interpreter line edit (the kernel specs will be created for *C:/Python39/python.exe* in the picture below). You can make kernel specs for other Pythons and virtual environments (venv) by changing the Python interpreter line edit path to point to another Python. Please see specific instructions for creating kernel specs for Conda environments below.



Once the **ipykernel** package is installed, the wizard runs the **ipykernel install** command, which creates the kernel specs directory on your system. Once the process finishes, click *Close*, and the newly created kernel spec (*python39* in this case) should be selected automatically. For convenience, there is a context-menu (mouse right-click menu) in the **Select Python Kernel…** combo box that opens the the kernel spec directory in your file browser. Click *Ok* to close the **Settings** widget and to save your selections.



If something went wrong, or if you want to remake the kernel specs, you can remove the kernel spec directory from your system, try the **Make Python Kernel** button again, or install the kernel specs manually.

If you want to install the Python kernel specs manually, these are the commands that you need to run.

To install **ipykernel** and it's dependencies, run:

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

And to install the kernel specs run:

```
python -m ipykernel install --user --name python39 --display-name python39_spinetoolbox
```

Make sure to use the `--user` argument to in order to make the kernel specs discoverable by Spine Toolbox.

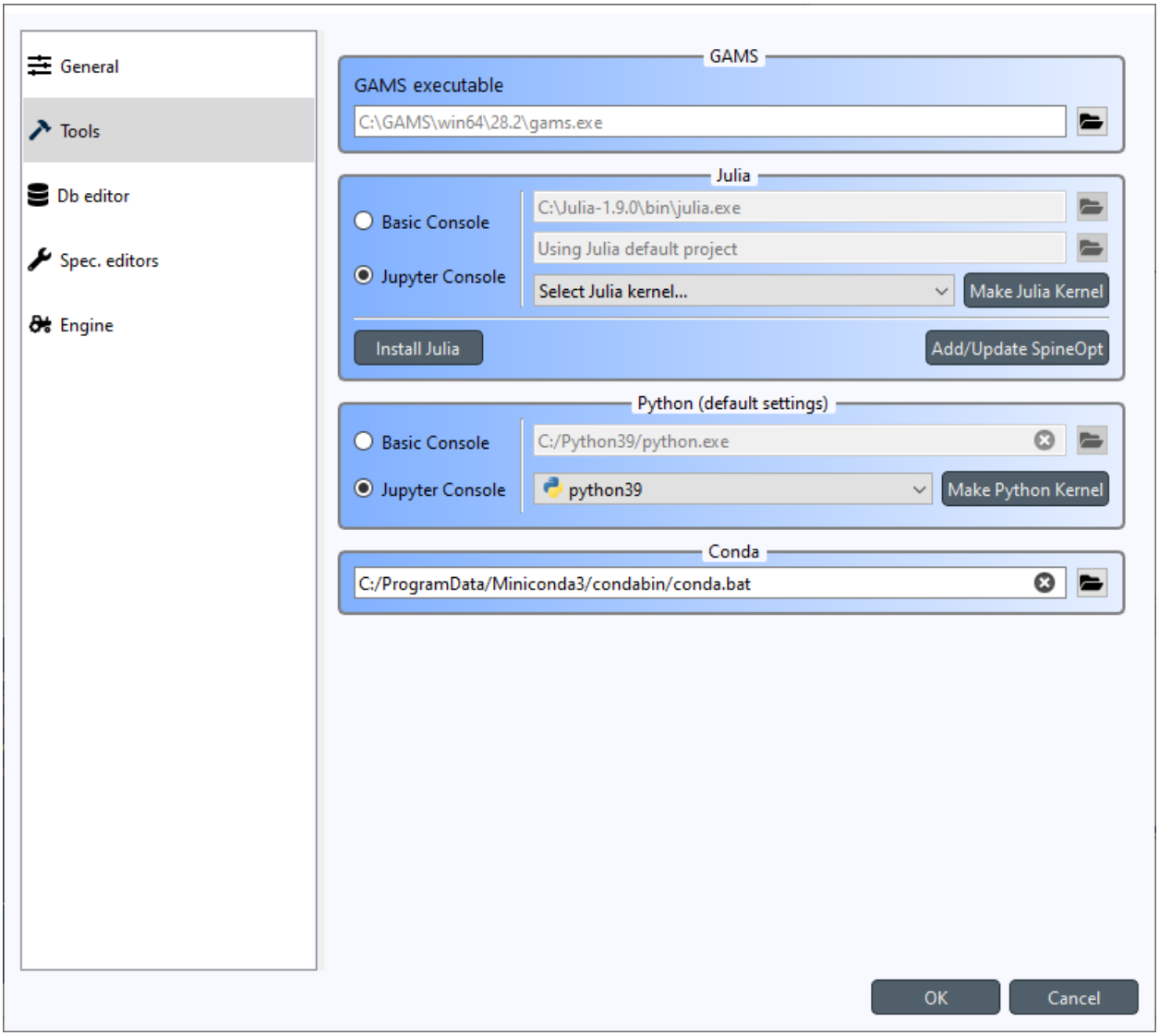
Important: If you want to have access to *spinedb_api*, you need to install it manually for the Python you select here.

Note: Clicking **Make Python Kernel** button when the kernel specs have already been installed, does NOT open the **Python Kernel Specification Creator**, but simply selects the Python kernel automatically.

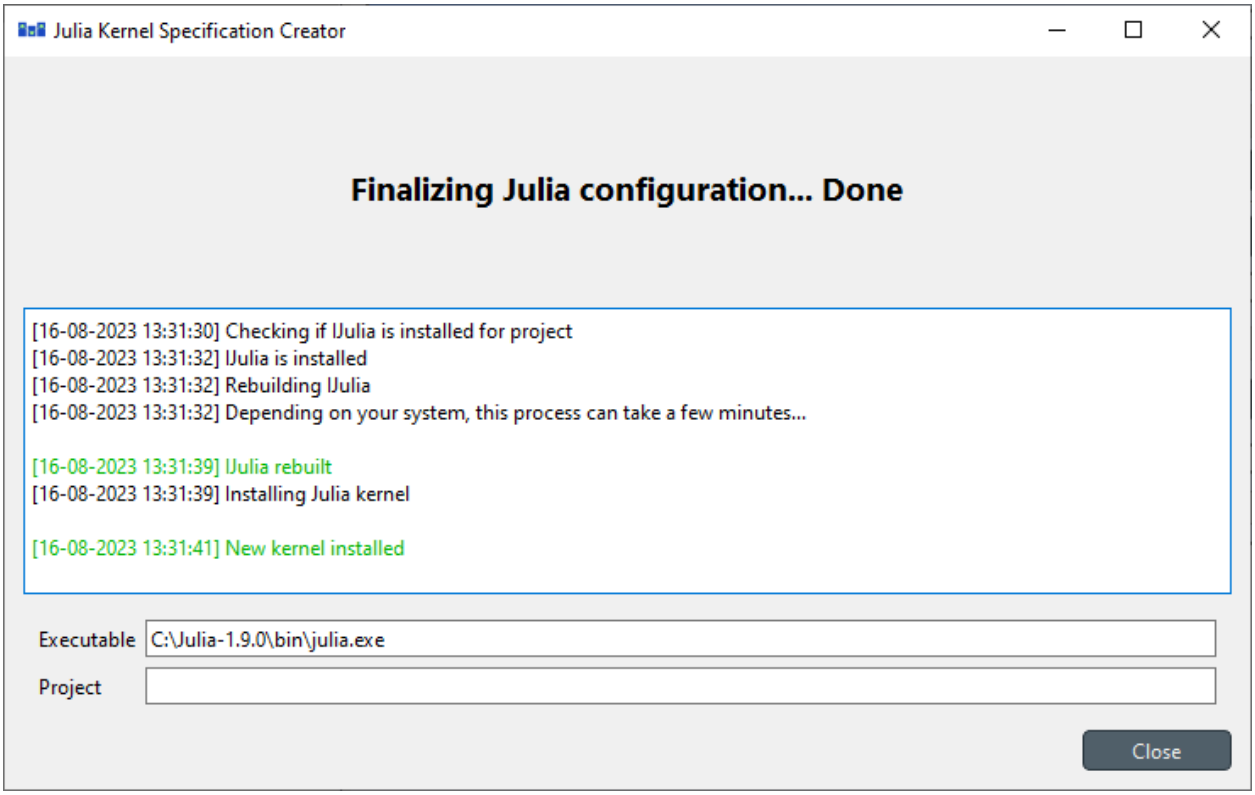
Note: Executing Python Tools using the Jupyter Console supports Python versions from 2.7 all the way to latest one. This means, that if you still have some old Python 2.7 scripts lying around, you can incorporate those into a Spine Toolbox project workflow and execute them without modifications.

4.2.2 Julia

To use the Jupyter Console with Julia Tools, go to the *Tools* page in **File -> Settings** and select the Jupyter Console radio button like in the picture below.



Like with Python, you need to select an existing Julia kernel for the Julia Jupyter Console, or create one either manually, or by clicking the **Make Julia Kernel** button.



Clicking the button opens **Julia Kernel Specification Creator** window, that first installs the **IJulia** package (if missing) for the Julia and Julia project that are currently selected in the Julia executable and Julia Project line edits (the kernel specs will be created for the default project of *C:/Julia-1.9.0/bin/julia.exe* in the picture above).

If something went wrong, or if you want to remake the kernel specs, you can remove the kernel spec directory from your system, try the **Make Julia Kernel** button again, or install the kernel specs manually.

If you want to install the Julia kernel specs manually, these are the commands that you need to run.

To install **IJulia** and it's dependencies, open Julia REPL with the project you want and run:

```
using Pkg
Pkg.add("IJulia")
```

Rebuild IJulia:

```
Pkg.build("IJulia")
```

And to install the kernel specs run:

```
using IJulia
installkernel("julia", --project="my_project")
```

Note: Clicking **Make Julia Kernel** button when the kernel specs have already been installed, does NOT open the **Julia Kernel Specification Creator**, but simply selects a Julia kernel that matches the selected Julia executable and Julia Project. If a kernel spec matching the Julia executable is found but the Julia project is different, a warning window appears, saying that Julia kernel spec may be overwritten if you continue.

4.2.3 Conda

You also have the option of running Python Tools in a Conda environment. All you need to do is the following.

1. Open Anaconda Prompt and make a new Conda environment:

```
conda create -n test python=3.10
```

2. Activate the environment:

```
conda activate test
```

3. Install **ipykernel**:

```
pip install ipykernel
```

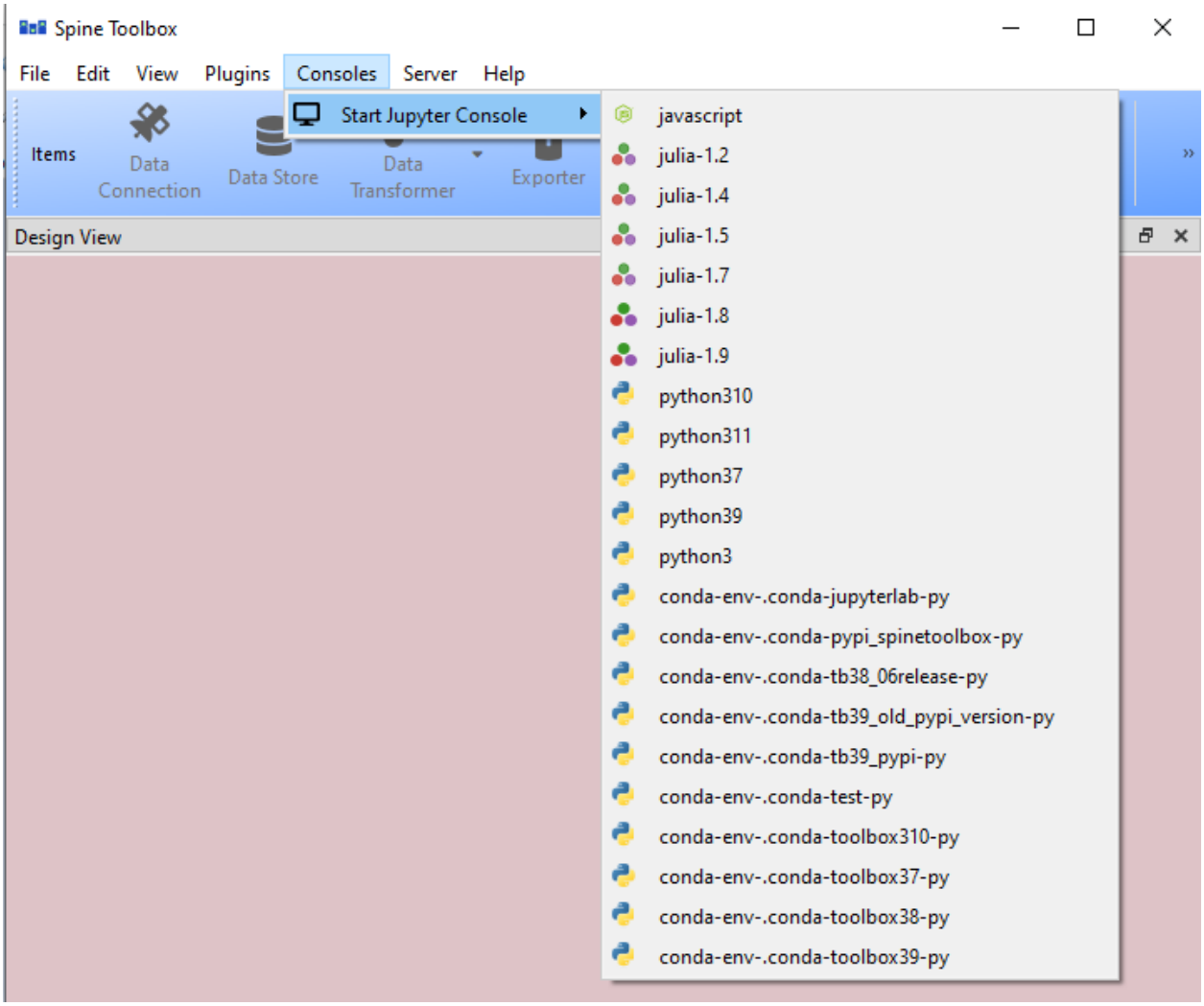
4. Back in Spine Toolbox, add path to Conda executable on the *Tools* page in **File -> Settings**.

That's it! Now, in Spine Toolbox main window, open the **Consoles -> Start Jupyter Console** menu, wait a second, and the new kernel should appear in the list. In this case, the new kernel name is *conda-env-.conda-test-py*. This autogenerated name will most likely change to something more readable in the future. You can use Conda Python kernels just like regular Python kernels, i.e. select one of them as the default kernel in the **File -> Settings** widget or select them for individual Python Tool Specs in **Tool Specification Editor** directly.

4.2.4 Detached Consoles

You can open 'detached' Jupyter Consoles from the main window menu **Consoles -> Start Jupyter Console**. The menu is populated dynamically with every Jupyter kernel that Spine Toolbox is able to find on your system. 'Detached' here means that the Consoles are not bound to any Tool. These Consoles are mostly useful e.g. for checking that the kernel has access to the correct packages, debugging, small coding, testing, etc. These may be especially useful for checking that everything works before running a full workflow that may take hours to finish.

Officially, Spine Toolbox only supports Python and Julia Jupyter kernels but it's possible that other kernels can be accessed in a Detached Console as well. For example, if you install a javascript kernel on your system, you can open a Detached Console for it, but this does not mean that Spine Toolbox projects should support Javascript. However, if there's interest and legitimate use cases for other kernels, we may build support for them in future releases.



If interested, you can read more on Jupyter kernels . There you can also find a list of available kernels.

4.3 GAMS

Executing Gams Tools or needing to use the GDX file format requires an installation of Gams on your system. You can download Gams from https://www.gams.com/download/.

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

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

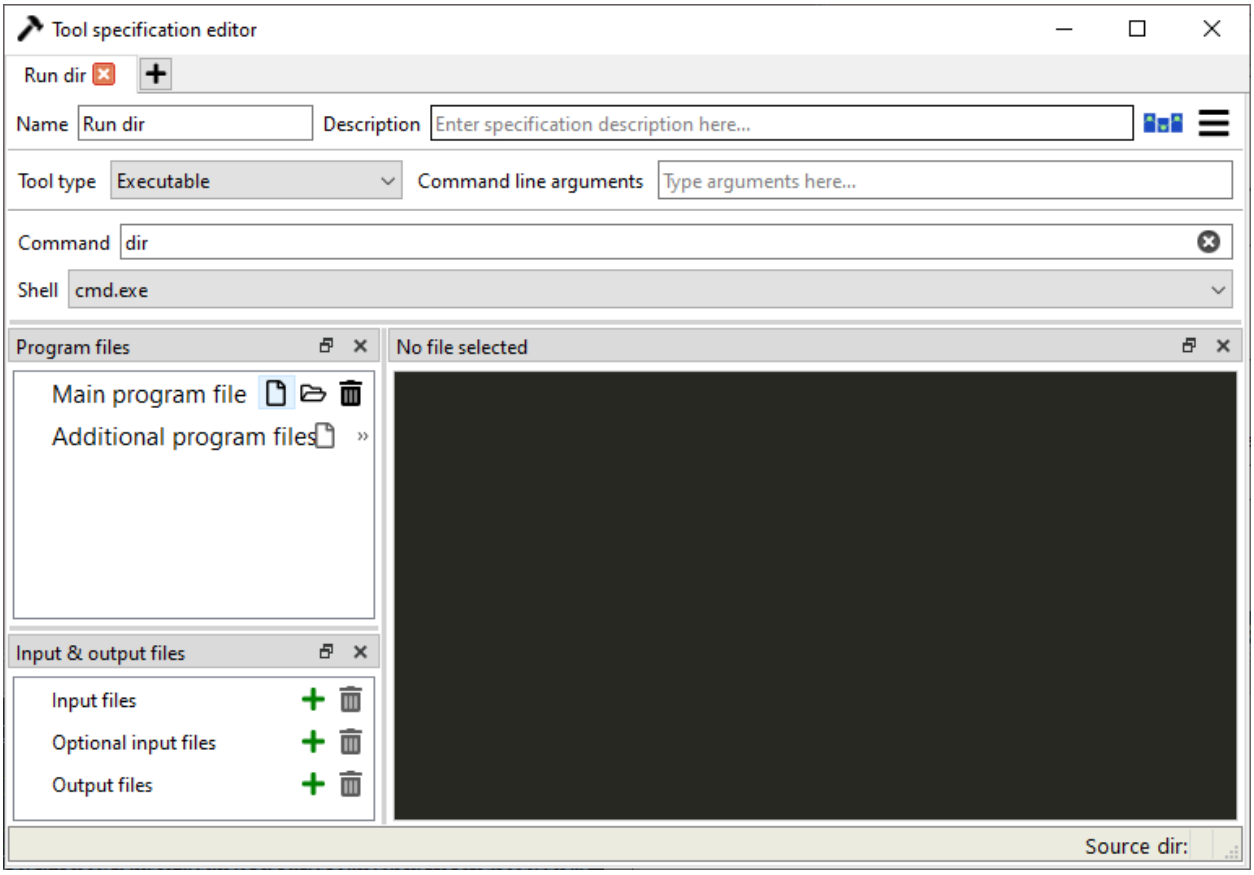
The default Gams is the Gams defined under `gams.location` in Windows registry or in your PATH environment variable. You can see the one that is currently in use from the *Tools* page in **File -> Settings**. The placeholder text shows the default Gams if found. You can also override the default Gams by setting some other gams executable path to the line edit.

4.4 Executable

Executable Tool Spec types can be used to execute virtually any program as part of a Spine Toolbox workflow. They also provide the possibility to run Shell commands as part the workflow. To run an executable with a shell you need to select a shell out of the three available options that is appropriate for your operating system. Then you can write a command that runs the executable with the arguments that it needs into the *Command* line edit just like you would on a normal shell.

To run an executable file without a shell you can either select the executable file as the main program file of the Tool and write the possible arguments into *Command line arguments* or select *no shell* and write the filepath of the executable file followed by it's arguments into the *Command* textbox. Either way the file is executed independent of a shell and with the provided arguments.

To run a Shell command, just type the command into the *command* line edit and select the appropriate Shell from the list. Picture below depicts an Executable Tool Spec that runs *dir* in in cmd.exe.



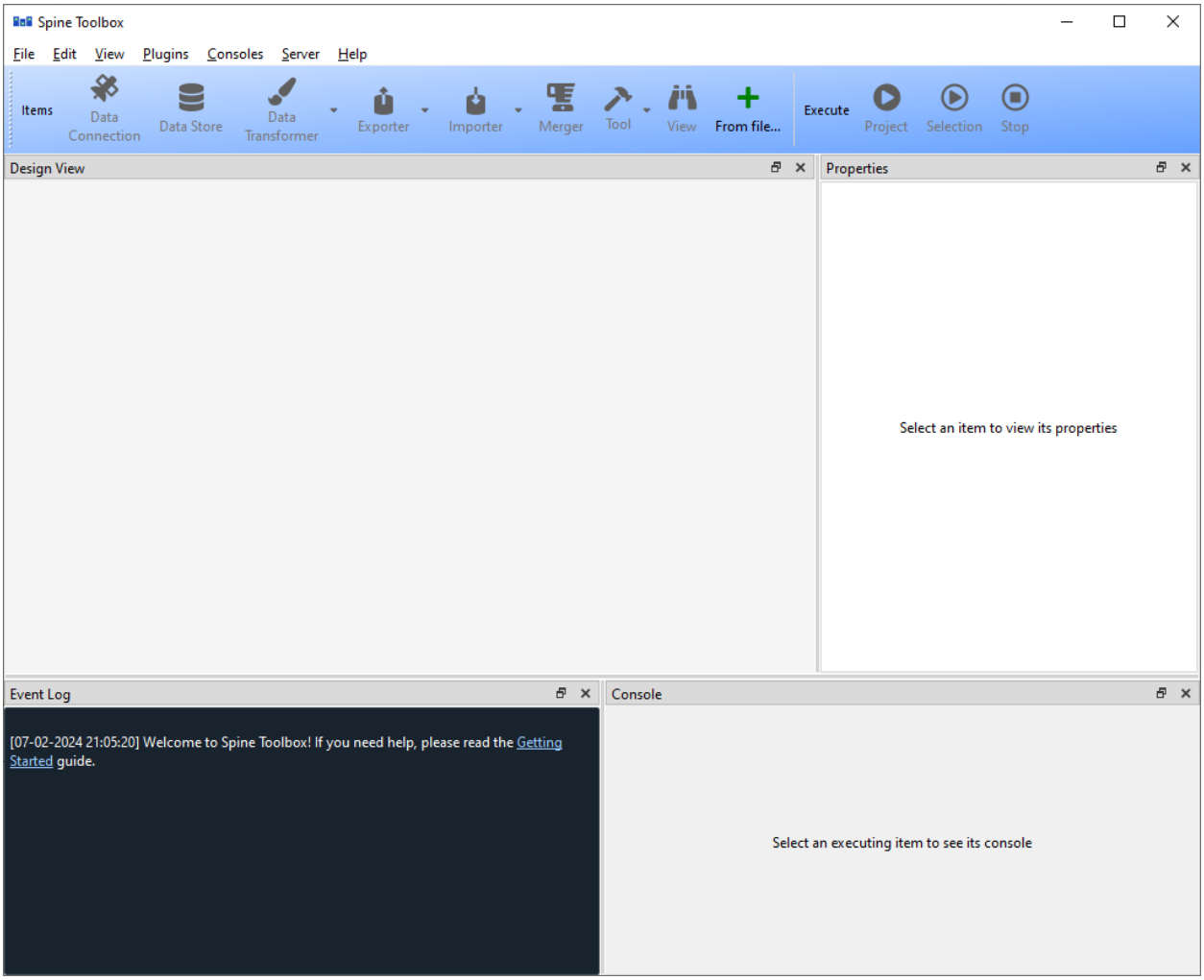
CHAPTER

FIVE

MAIN WINDOW

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

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



The application main window contains four dock widgets (**Design View**, **Properties**, **Event Log**, **Console**), a **Toolbar**, and a menu bar with **File**, **Edit**, **View**, **Plugins**, **Consoles**, **Server** and **Help** menus. The **Properties** dock widget shows the properties of the selected project item. **Event Log** shows messages based on user actions and item executions. It shows messages from processes that are spawned by the application, i.e. it shows the stdout and stderr streams of executable programs. In addition, it displays messages related to item's execution.

Console provides Julia and Python consoles that can be interacted with. What kind of console is shown depends on the Tool type of the specific Tool. Only an item that is currently executing or has already executed shows something in this dock widget. To view an item's console, the item must be selected. When executing Python/Julia tools, the Tool's Python/Julia code will be included into the console and executed there.

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**, **Plugins**, **Consoles**, **Server** and **Help** menus. In the **File** menu you can create a new project, open an existing project, save the project or 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. In the **Edit** menu you can for example copy, paste and duplicate items as well as undo and redo actions. In the **Plugins** menu you can install and manage plugins. **Consoles** menu provides a way to start detached consoles. In the **Server** menu you can retrieve projects from a server. **Help** contains a link to this documentation as well as various tidbits about Spine Toolbox.

The **Items** section of the **Toolbar** contains the available *project item* types. The **Execute** section contains icons that control the execution of the items in the **Design view**. The button executes all workflows in the project in parallel. The button executes the selected project items only. The button terminates the execution (if running).

You can add a new project item to your project by pointing your mouse cursor on any of the draggable items in the **Toolbar**, 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, rearrange the order of the dock widgets by dragging them around and/or resize the views to fit your needs and display size or resolution. You can find more ways to customize the visual elements of Spine Toolbox in the *settings*.

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.

CHAPTER

SIX

PROJECT ITEMS

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

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

6.1 Project Item Properties

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

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

6.2 Project Item Descriptions

The following items are currently available:

6.2.1 Data Connection

A Data connection item provides access to data files. The item has two categories of files: **references** connect to files anywhere on the file system or on remote (non-Spine) databases while **data** files reside in the item's own data directory.

6.2.2 Data Store

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

6.2.3 Data Transformer

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

6.2.4 Exporter

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

6.2.5 Importer

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

6.2.6 Merger

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

6.2.7 Tool

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

6.2.8 View

A View item is meant for plotting preselected parameter values from multiple sources.

CHAPTER

SEVEN

LINKS

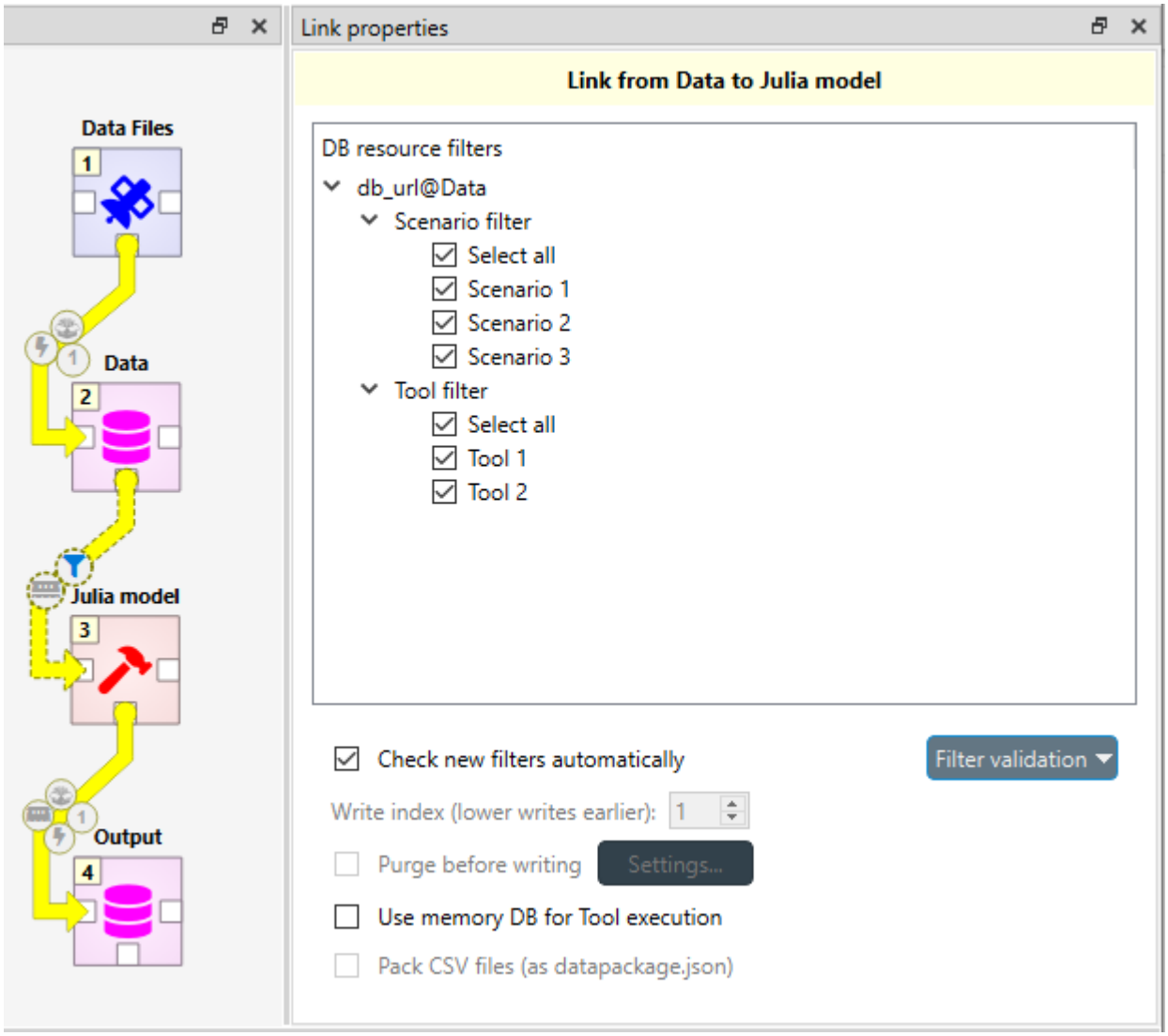
Links are the things that connect project items to each other. If Tool is the heart of a DAG, then links are the veins that connect the heart to other vital organs.

Creating a new link between items is simple. First you need to select any of the connector slots on the item where you want the link to originate form. Then select any connector slot on the item that you want to connect to. There are no limitations for how many links one connector slot can have. Like items, links can also have properties, depending on the types of items that they are connecting. When a link is selected, these properties can be modified in the **Properties** dock widget.

The small bubble icons in a link represent the state of the link's properties. When an icon is blue, the corresponding selection is active in the **Properties** dock widget.

7.1 Data Store as Source

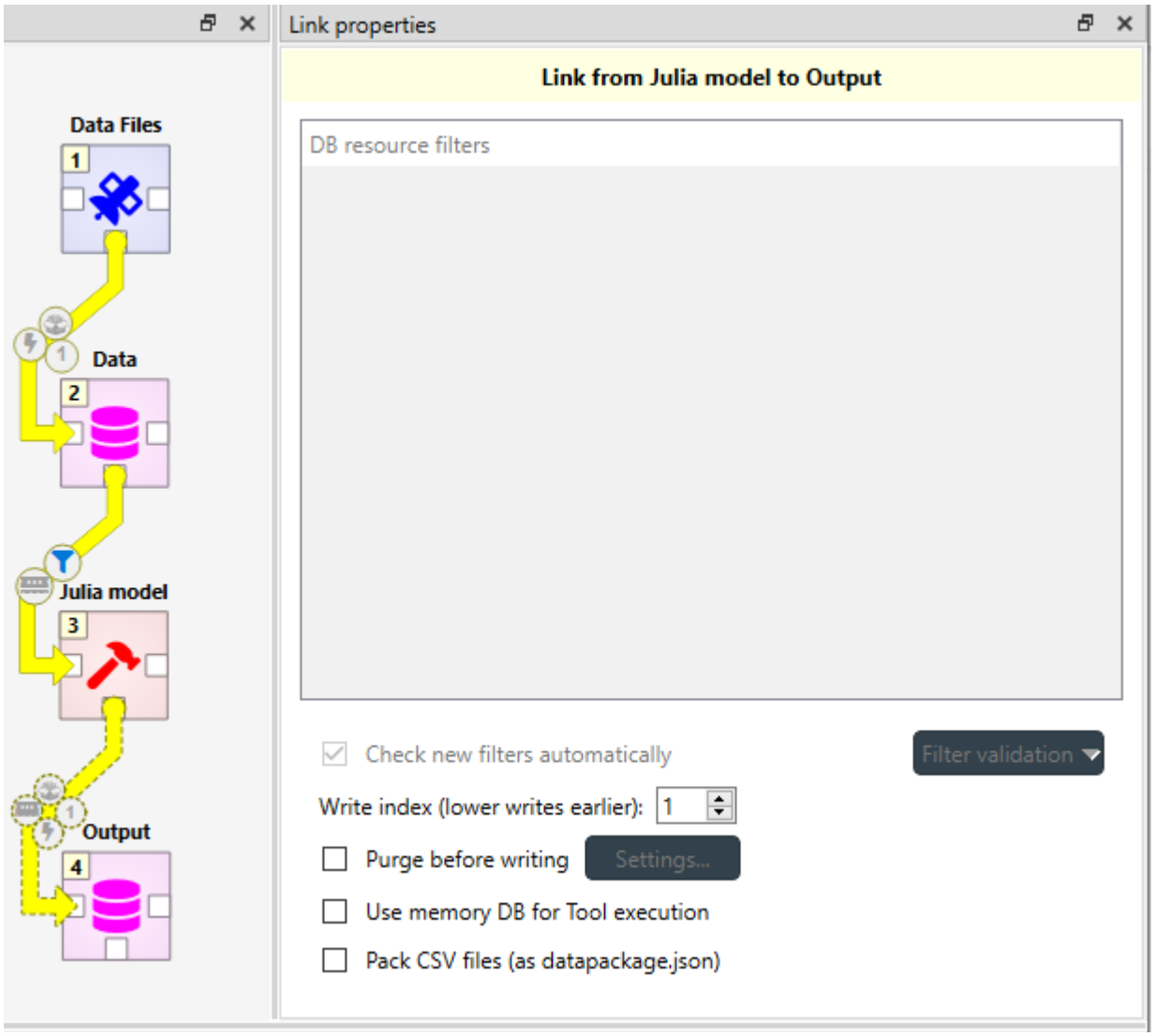
Below is an example of what the **Properties** dock widget can look like when a link originating from a Data Store is selected:



DB resource filters is divided into *Scenario* and *Tool filters*. With the scenario filters, you can select which scenarios to include in the execution. The *Tool filter* lets you choose which tools specified in the database are active. The *Check new filters automatically* option allows you to choose whether new *Scenario* and *Tool filters* added to the database should be automatically selected in this specific link. *Filter validation* allows you to force that at least one *Scenario* and/or *Tool filter* is selected at all times in that specific link.

7.2 Data Store as Destination

In the image below, the selected link ends in a Data Store. Because of this, the available selections in **Properties** differ from the previous image.



Now the link has no filters, but the write index and other various database related options become available.

Write index controls which items write to the database first. Smaller indices take precedence over larger ones while items with the same index write in an undefined order.

Purge before writing option purges the target Data Store before write operations. Click on **Settings...** to set up which items to purge.

Warning: This purge has no undo available.

7.3 Using Memory Databases

Use memory DB for Tool execution allows using a temporary in-memory database while executing a Tool which may speed up execution if the Tool accesses the database a lot.

7.4 Packing CSV files into datapackage

When the source item may provide output files, the **Pack CSV files (as datapackage.json)** option becomes enabled. This option may be handy when an item provides a lot of CSV files that e.g. need to imported into a Data Store. Checking this options does two things:

- A `datapackage.json` file is created in the common parent directory of all CSV files the source item provides. This file defines a datapackage that consists of the CSV files.
- The destination item receives only the `datapackage.json` file instead of any CSV files from the source item.

See the datapackage specification for more information on datapackages.

CHAPTER

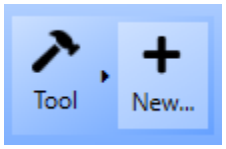
EIGHT

TOOL SPECIFICATION EDITOR

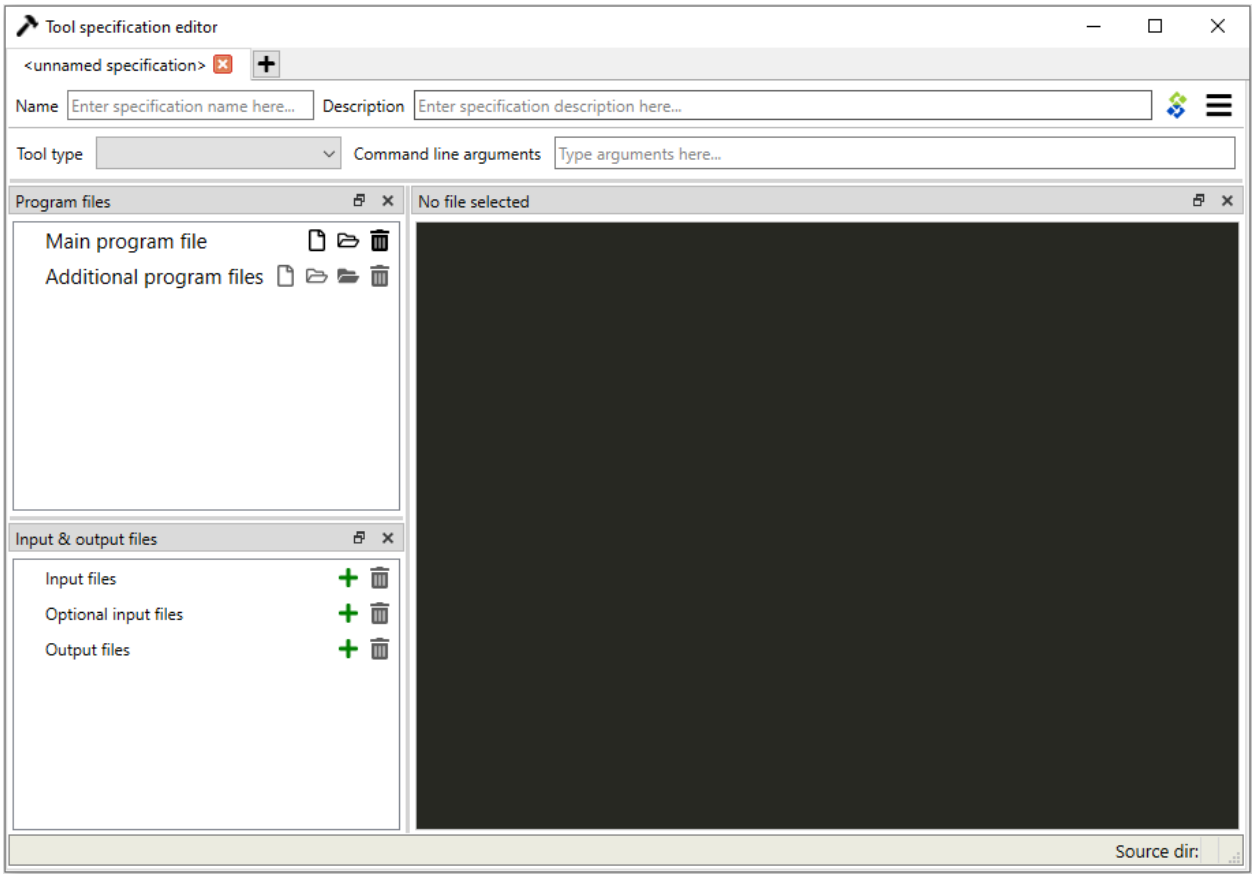
This section describes how to make a new Tool specification and how to edit existing Tool specifications.

8.1 General

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



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



Start by giving the Tool specification a name. Then select the type of the Tool. You have four options (Julia, Python, GAMS or Executable). You can give the Tool specification a description, describing what the Tool specification does. Main program file is the main file of your tool, i.e. a script that can be passed to Julia, Python, GAMS, or the system shell. You can create a blank file by pressing the button, or you can browse to find an existing main program file by pressing the button.

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

Tip: Another way to pass arguments to a Tool is to write them into the *Tool arguments* drop-down list in the **Properties** dock widget. There it is possible to also rearrange existing arguments or to select available resources that are provided by other project items as arguments.

Unlike the arguments set in Tool Specification Editor, the arguments in **Properties** are *Tool specific*.

Additional source files is a list of files that the main program requires in order to run. You can add individual files the same way as with the main program file or whole directories at once by pressing the button.

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 inside Tool's data directory after the Tool specification has finished execution. Unix-style wildcards ? and * are supported.

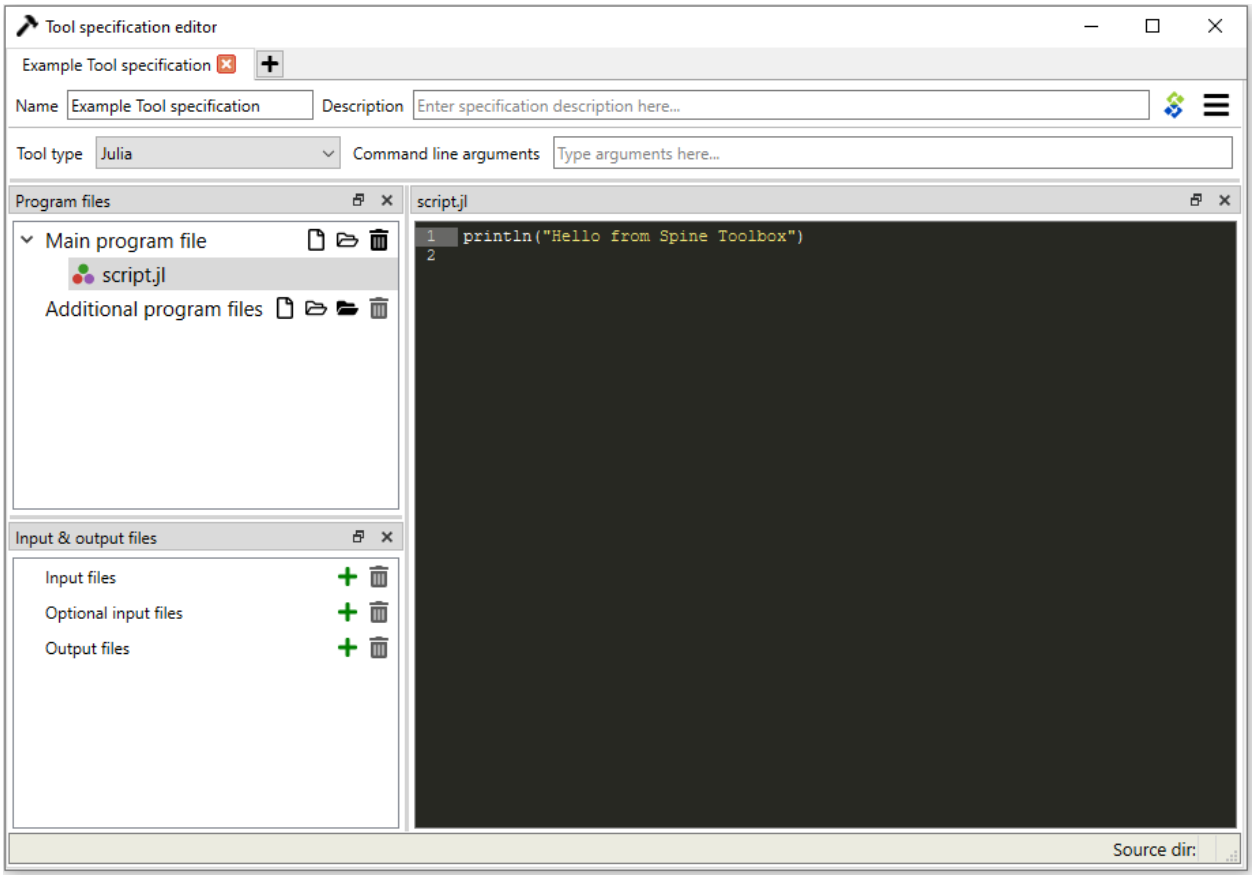
Examples:

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

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

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

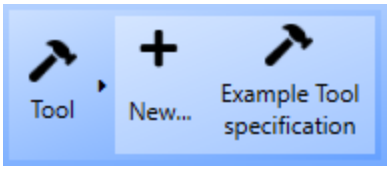
Here is a minimal Tool specification for a Julia script *script.jl*



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

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

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



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

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

8.2 Input & Output Files in Practice

The file names can be either hard coded or not. For example, you could have a script that requires (hard coded in the script) a file *input.dat* and optionally works with a bunch of files that are expected to have the *.csv* extension. In that case you would define

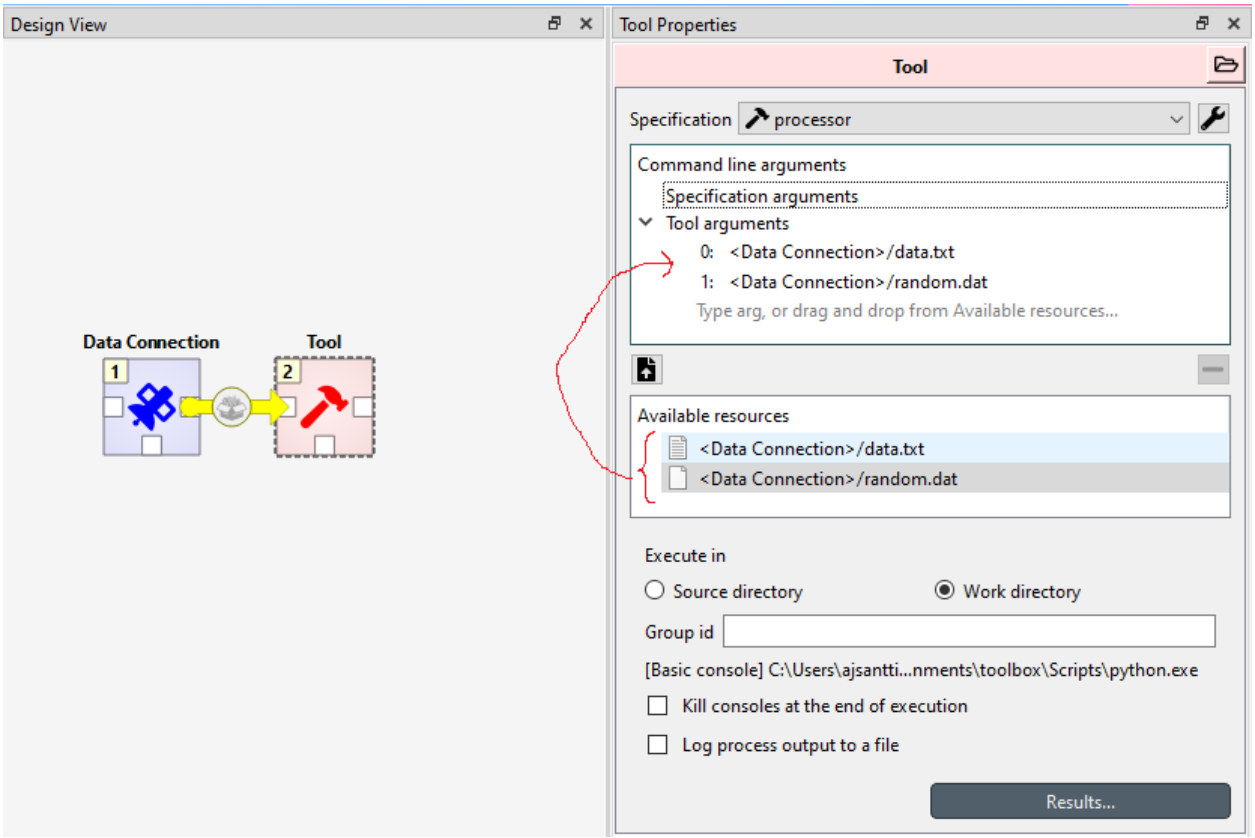
- *input.dat* as an Input file
- **.csv* as Optional input files

The *Output files* work similarly; you can hard code the entire file name or use wildcards for *Optional output files*.

When specifying the *Input* and *Output files* in the Specification editor, Toolbox will copy the files to the Tool's work directory when the Tool is executed, so they are available for the script in a known location. Note, that you can specify subdirectories for the files as well. These will be relative to the work directory.

These options expect some level of hard-coding: file names, or at least file extensions as well as relative locations to the work directory need to be known when writing the Tool Spec script.

There is another, more general way to provide *Input files* to the script that does not require any kind of hard coding: *command line arguments*. You can set them up in **Tool's Properties** tab. For example, in the project below, a Data connection provides *Input files* for the workflow. The files are visible in the *Available resources list* in **Tool's Properties** and they have been *dragged and dropped* into the the Tool arguments list.



Now, the Python script can access the files using something like:

```
import sys
file_path1 = sys.argv[1]
file_path2 = sys.argv[2]
```

Of course, more serious scripts would use the *argparse* module.

CHAPTER

NINE

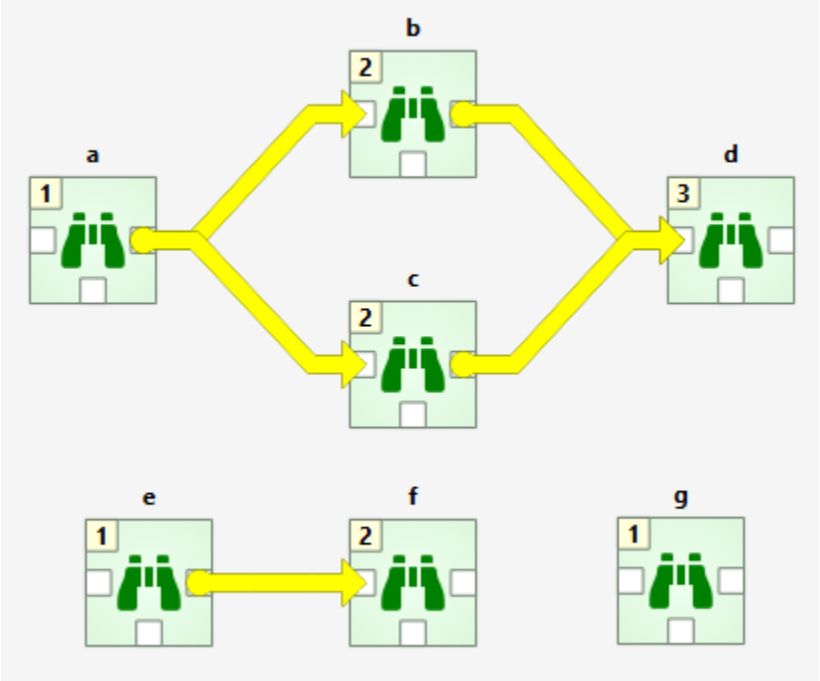
EXECUTING PROJECTS

This section describes how executing a project works and what resources are passed between project items at execution time. The buttons used to control executions are located in the **Toolbar**'s Execute -section. Execution happens by either pressing the *Execute Project* button () to execute the whole project, or by pressing the *Execute Selected* button () to only execute selected items. Next to these buttons is the **Stop** button (), which can be used to stop an ongoing execution. 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*. The DAG is traversed using the **breadth-first-search** algorithm.

Rules of DAGs:

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

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



- DAG 1: items: *a*, *b*, *c*, *d*. connections: *a*-*b*, *a*-*c*, *b*-*d*, *c*-*d*

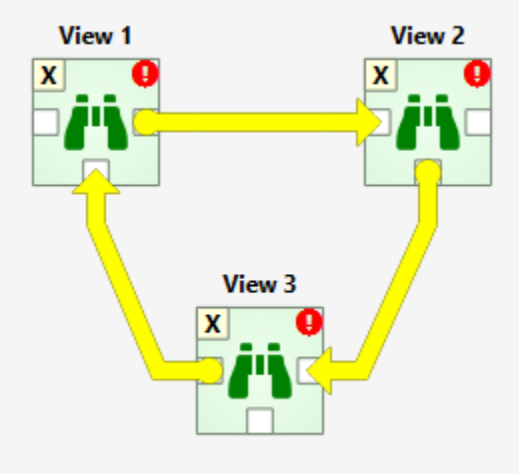
- DAG 2: items: *e*, *f*. connections: *e-f*
- DAG 3: items: *g*. connections: None

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

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

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

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

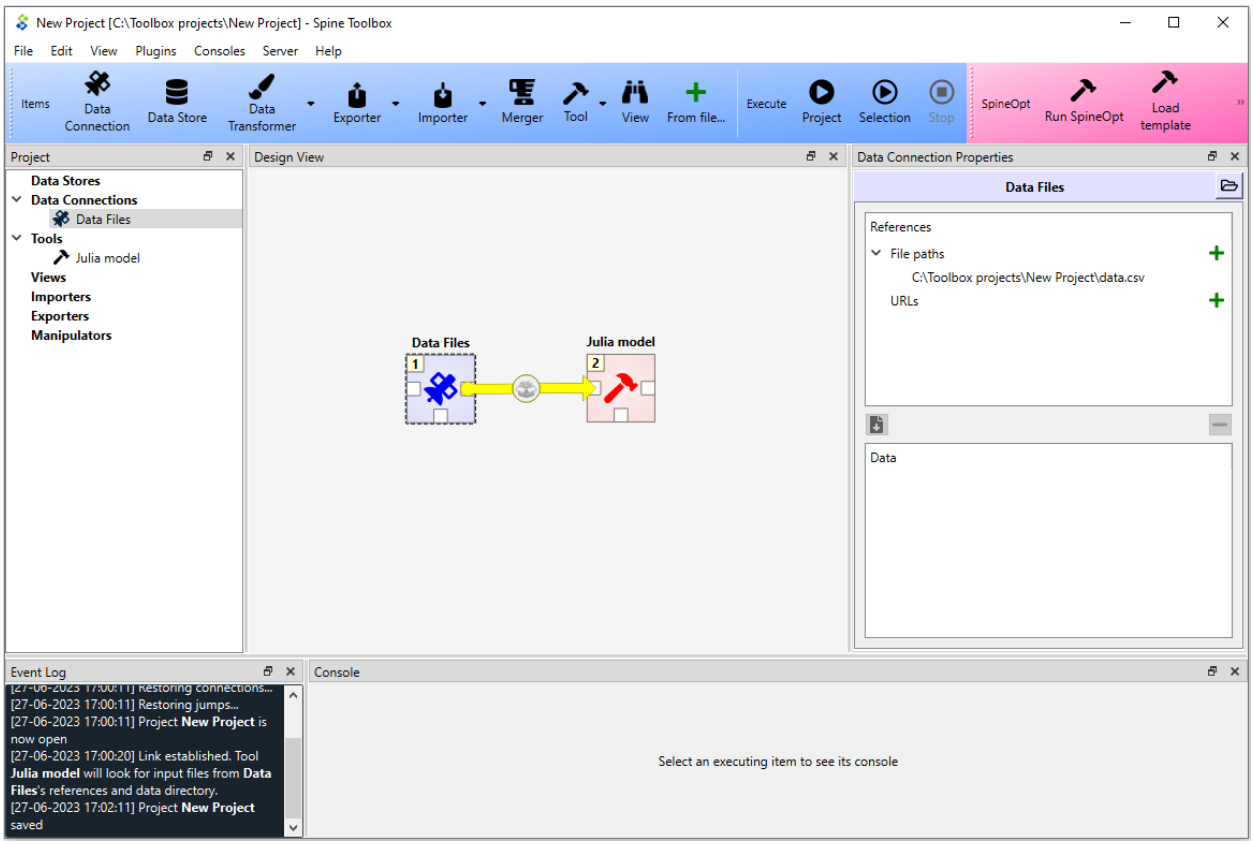


You can also execute only the selected parts of a project by multi-selecting the items you want to execute and pressing the button in the tool bar. For example, to execute only items *b*, *d* and *f*, select the items in **Design View** button.

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

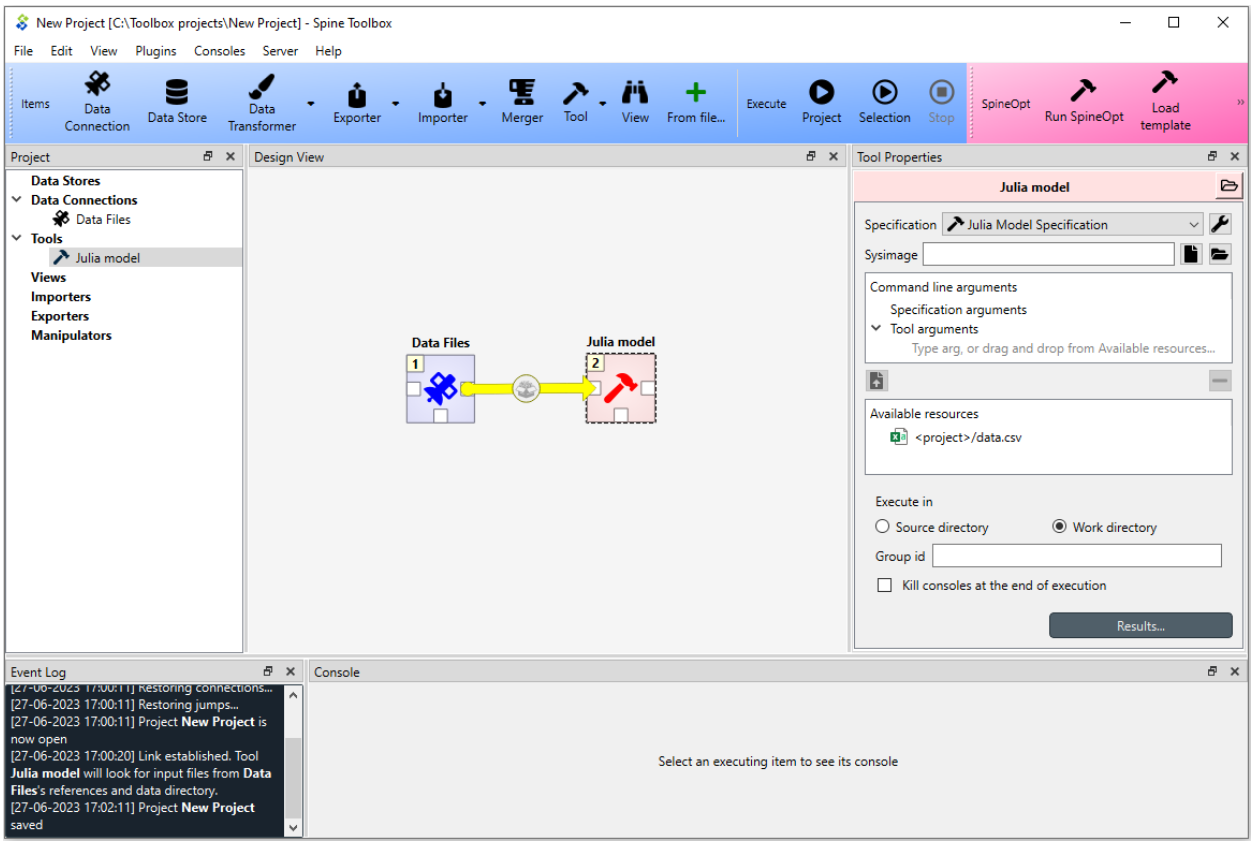
9.1 Example DAG

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



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

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



In the top of the Tool Properties, there is **Specification** drop-down menu. From this drop-down menu, you can select the Tool specification for this particular Tool item. The *Julia Model Specification* tool specification has been selected for *Julia model*. Below the drop-down menu, you can choose a precompiled sysimage and edit Tool's command line arguments. Note that the command line argument editor already 'sees' the `data.csv` file provided by *Data Files*. 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. In Group id, an execution group identifier can be given. Below that, there is a checkbox with the choice to kill consoles after execution. *Results...* button opens the Tool's result archive directory in system's file browser (all Tools have their own result directory).

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

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

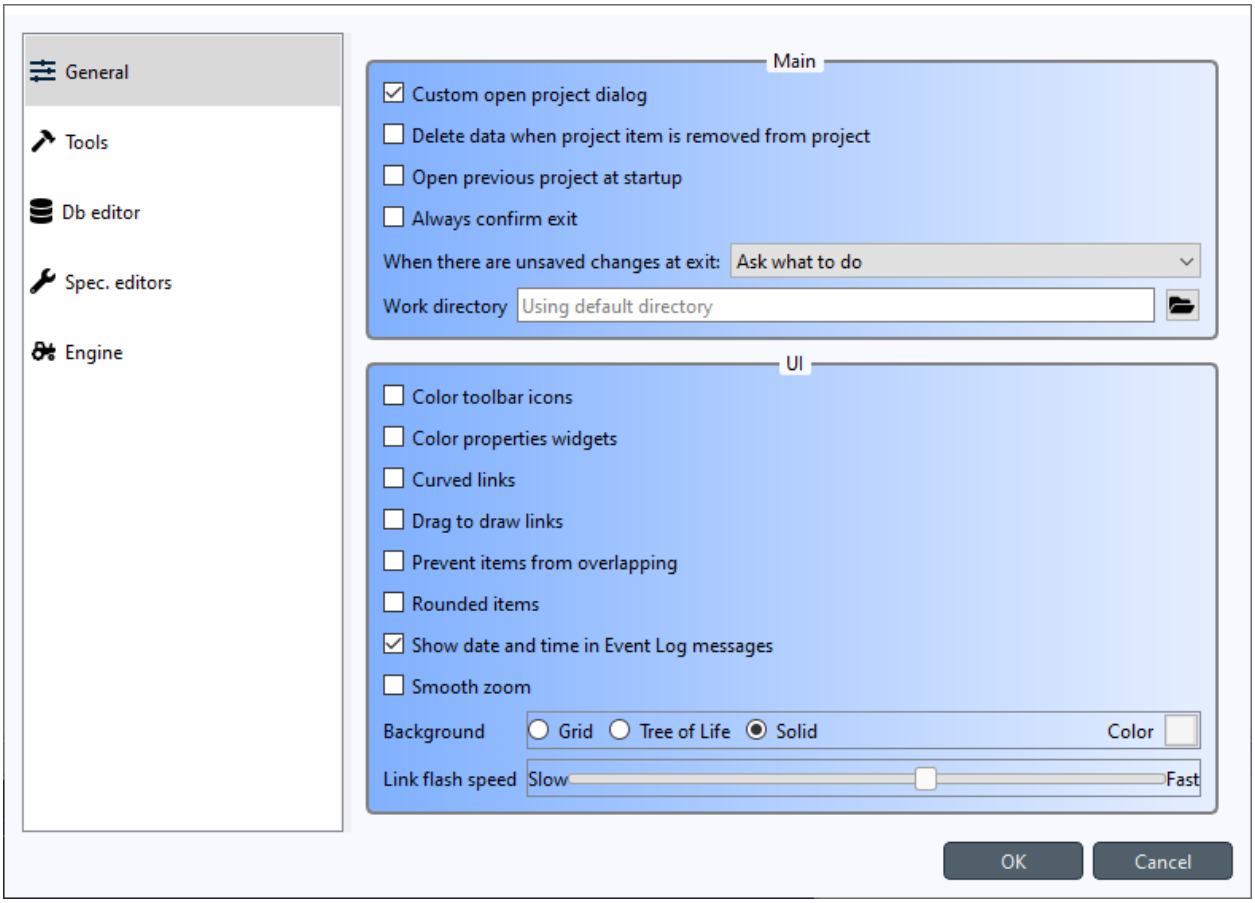
CHAPTER

TEN

SETTINGS

You can open Spine Toolbox settings from the main window menu **File -> Settings...**, or by pressing **Ctrl+,**. Settings are categorized into five tabs; *General*, *Tools*, *Db editor*, *Spec. editors* and *Engine*. In addition to application settings, each project item has user adjustable properties (See *Project Items*). See also *Setting up Consoles and External Tools* for more information on how to set up Consoles and external Tools.

10.1 General Settings



The General tab contains the general application settings.

Settings in the **Main** group:

- **Custom open project dialog** If checked, the application uses a special-purpose dialog when opening a project. If left unchecked, the operating system's default dialog is used.
- **Delete data when project item is removed from project** Check this box to delete project item's data when a project item is removed from project. This means that the *project item directory* and its contents will be deleted from your hard drive. You can find the project item directories from the *<proj_dir>/.spinetoolbox/items/* directory, where *<proj_dir>* is your current project directory.
- **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.
- **Always confirm exit** If checked, confirm exit prompt is shown always. If unchecked, application exits without prompt when there are no unsaved changes.
- **When there are unsaved changes at exit** The combo box chooses what to do with unsaved changes when the application exits.
- **Work directory** Directory where processing Tools takes place. Default place (if left empty) is shown as placeholder text. Make sure to clean up the directory every now and then.

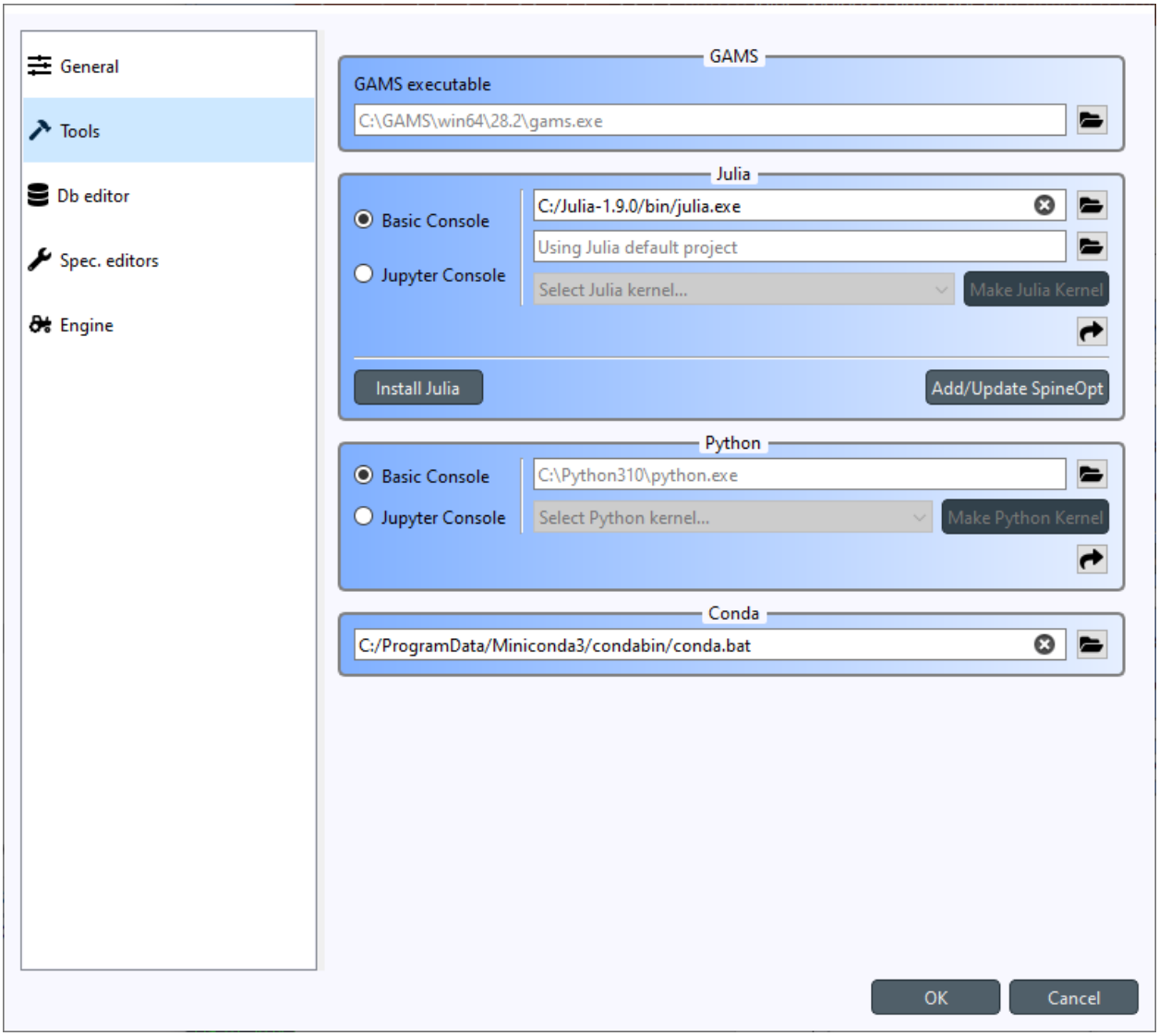
Settings in the **UI** group:

- **Color toolbar icons** Check this box to give some color to the otherwise black toolbar icons.

- **Color properties widgets** Check this box to make the background of Project item properties more colorful.
- **Curved links** Controls the look of the arrows on **Design View**.
- **Drag to draw links** When checked, the mouse button needs to be pressed while drawing links between project items. If unchecked, single clicks at link source and destination connector slots suffices.
- **Prevent items from overlapping** When checked, other project items can be pushed away when moving an item around the **Design view**. If left unchecked, items can be piled on top of each other.
- **Rounded items** Check this box to round the corners of otherwise rectangular project items.
- **Show date and time in Event Log messages** If checked, every **Event Log** message is prepended with a date and time 'tag'.
- **Smooth zoom** Controls the way zooming (by using the mouse wheel) behaves in **Design View** and in **Spine DB Editor**. Controls if the zoom in/out is continuous or discrete. On older computers, smooth zoom is not recommended because it may be slower.
- **Background** Has some pattern options for the background of the **Design View**. Clicking on the square next to 'Color' lets you choose the pattern's color.
- **Link flash speed** This slider controls the speed of the link animation on **Design View** when execution is ongoing.

10.2 Tools Settings

The Tools tab contains settings for external tools.



Settings in the **GAMS** group:

- **GAMS executable** Set the path to GAMS executable you want to use when executing GAMS tools. If you have GAMS in your PATH environment variable, it will be automatically used. You can also choose another GAMS by clicking the button.

Settings in the **Julia** group:

Choose the settings on how Julia Tools are executed.

- **Basic Console** When selected, Julia Tools will be executed in a custom interactive Julia REPL.
- **Julia executable** Set the path to a Julia Executable used in launching the Basic Console. If Julia is in PATH this will be autofilled, but you can also choose another Julia executable.
- **Julia project** Set the Julia project you want to activate in the Basic Console.
- **Jupyter Console** Choosing this option runs Julia Tools in a custom Jupyter QtConsole embedded into Spine Toolbox.
- **Select Julia kernel... drop-down menu** Select the kernel you want to launch in Jupyter Console.

Note: The Julia settings above are **the default settings for new Julia Tool Specs**. You can select a specific Julia executable & project or Julia kernel for each Tool Spec separately using the **Tool Specification Editor**.

- **Make Julia Kernel** Clicking this button makes a new kernel based on the selected *Julia executable*, and *Julia project*. The progress of the operation is shown in another dialog. Installing a Julia kernel requires the **IJulia** package which will be installed to the selected *Julia project*. After **IJulia** has been installed, the kernel is installed. This process can take a couple of minutes to finish.
- This button sets **all Julia Tools** execution settings in the current project to defaults. This operation is irreversible because the project will be saved afterwards.
- **Install Julia** Installs the latest Julia on your system using the **jill** package.
- **Add/Update SpineOpt** Installs the latest compatible **SpineOpt** to the selected Julia project. If the selected *Julia project* already has SpineOpt, it is upgraded if there is a new version available.

Settings in the **Python** group:

Choose the settings on how Python Tools are executed.

- **Basic Console** When selected, Python Tools will be executed in a custom interactive Python REPL.
- **Python executable** Set the path to a Python Executable used in launching the Basic Console. The default option (if the line edit is blank) is the Python executable that was used in launching Spine Toolbox.
- **Jupyter Console** Choosing this option runs Python Tools in a custom Jupyter QtConsole embedded into Spine Toolbox.
- **Select Python kernel… drop-down menu** Select the kernel you want to launch in Jupyter Console.

Note: The Python settings above are **the default settings for new Python Tool Specs**. You can select a specific Python executable or Python kernel for each Python Tool Spec separately using the **Tool Specification Editor**.

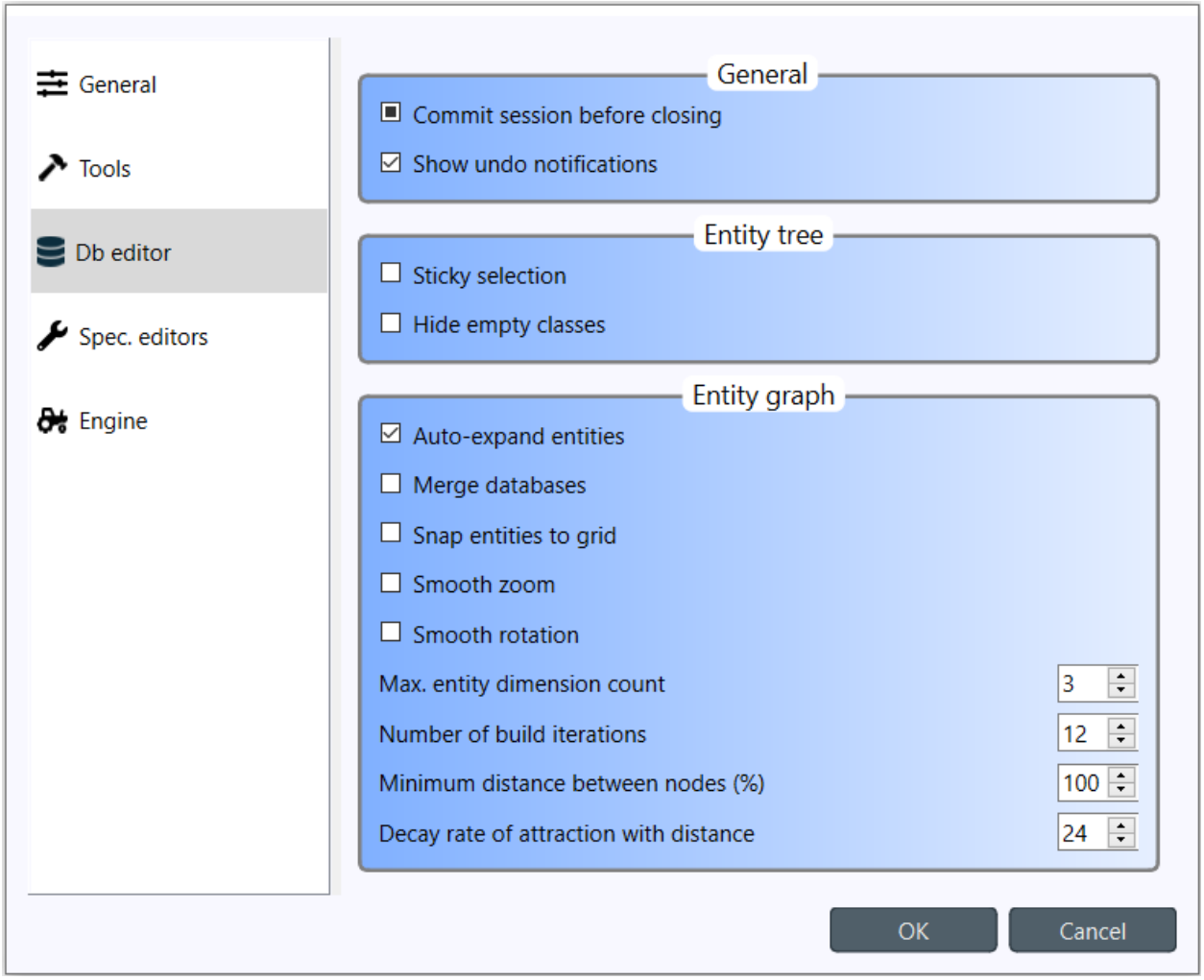
- **Make Python Kernel** clicking this button makes a new kernel based on the selected *Python executable*. The progress of the operation is shown in another dialog. Installing a Python kernel (actually IPython kernel) requires the **ipykernel** package which will be installed to the selected *Python executables*. After **ipykernel** has been installed, the kernel is installed. This process can take a couple of minutes to finish.
- This button sets **all Python Tools** execution settings in the current project to defaults. This operation is irreversible because the project will be saved afterwards.

Settings in the **Conda** group:

- **Miniconda executable** If you want to run Python Tools in a Conda environment, you can set the path to your Conda executable here.

See *Setting up Consoles and External Tools* for more information and examples.

10.3 Db editor Settings



This tab contains settings for the Spine Database editor. The same settings can be accessed directly from the Database editor itself.

Settings in the **General** group:

- **Commit session before closing** This checkbox controls what happens when you close a database editor which has uncommitted changes. When this is unchecked, all changes are discarded without notice. When this is partially checked (default), a message box warning you about uncommitted changes is shown. When this is checked, a commit message box is shown immediately without first showing the message box.
- **Show undo notifications** Checking this will show undo notification boxes in the editor every time something undoable happens. Unchecking hides the notifications.

Settings in the **Entity tree** group:

- **Sticky selection in entity trees** Controls how selecting items in Spine database editor's **Entity Tree** using the left mouse button works. If checked, multiple selection is enabled and pressing **Ctrl** enables single selection. If unchecked, single selection is enabled and pressing **Ctrl** enables multiple selection.

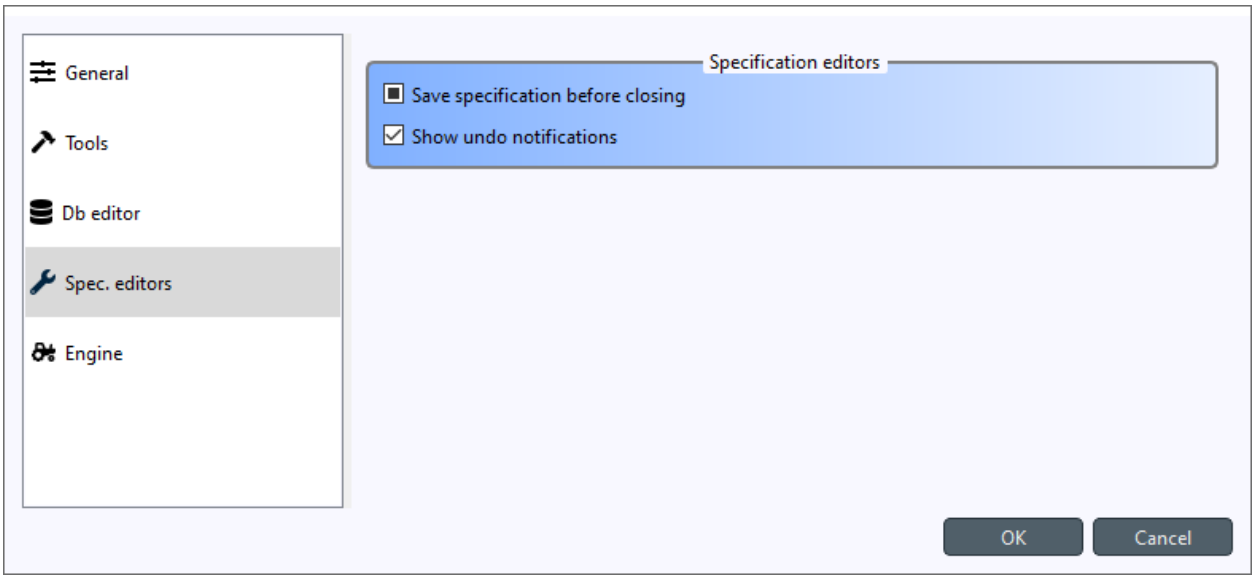
Settings in the **Entity graph** group:

- **Auto-expand entities** This checkbox controls which N-D entities are automatically shown on the Graph view.

If checked, all N-D entities that are related to the selection are included automatically. If unchecked, only N-D entities that have all the elements visible in the graph are shown automatically.

- **Merge databases** If checked, Graph view will combine all databases that are open on the same table into a single graph if they contains common entity nodes. If unchecked, a separate graph will be drawn for each database.
- **Snap entities to grid** Makes it so that the placement of the entities can't be arbitrary anymore but instead they can only lay on a grid.
- **Smooth zoom** Checking this enables smooth zoom on the Graph view.
- **Smooth rotation** Checking this enables smooth rotation on the Graph view.
- **Max. entity dimension count** Defines a cutoff for the number of dimensions an entity can have and still be drawn.
- **Number of build iterations** Defines the maximum numbers of iterations the layout generation algorithm can make.
- **Minimum distance between nodes (%)** Used for setting the ideal distance between entities in the graph.
- **Decay rate of attraction with distance** The higher this number, the lesser the attraction between distant vertices when drawing the graph.

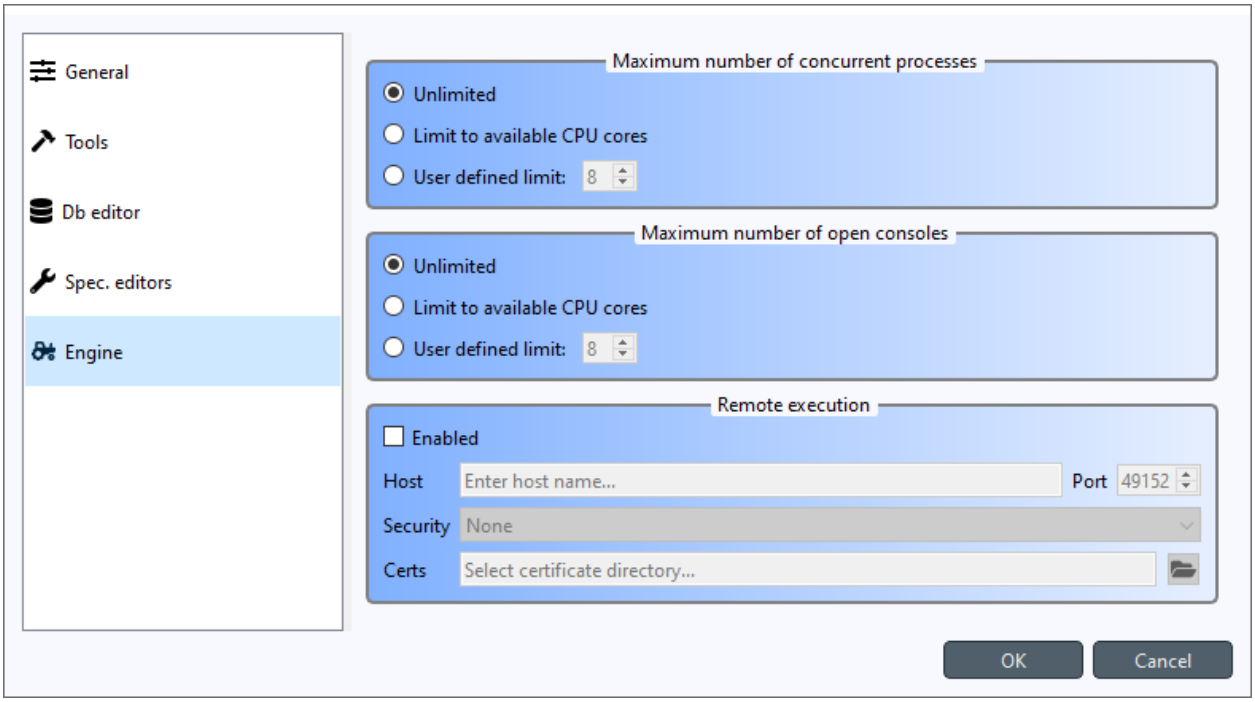
10.4 Spec. editor Settings



The Spec. editor tab contains common settings for all specification editors.

- **Save specification before closing** If checked, specification editors will save the specification automatically at exit. If partially checked, the editors will prompt what to do explicitly. If unchecked, no prompts will be shown and all changes will be lost at exit.
- **Show undo notifications** Checking this will show undo notification boxes in the editor every time something undoable happens. Unchecking hides the notifications.

10.5 Engine settings



The Engine settings tab contains settings for Spine Engine.

- **Maximum number of concurrent processes** This sets a limit on how many concurrent processes the Engine can launch. *Unlimited* means that there is no upper limit. With no limits to concurrent processes the execution never stalls waiting for processes to finish. It may, however, consume all system's resources. *Limit to available CPU cores* sets the upper limit to the number of cores on the system. Finally, exact upper limit can be set by the *User defined limit* spin box.
- **Maximum number of open consoles** This sets a limit on how many concurrent Python or Julia consoles (Basic and Jupyter) there can be running at the same time. Note, that this is a separate limit from the number of concurrent processes above. *Unlimited* means that there is no upper limit. With no limits to open consoles the execution never stalls waiting for console to become free. It may, however, consume all system's resources. *Limit to available CPU cores* sets the upper limit to the number of cores on the system. Finally, exact upper limit can be set by the *User defined limit* spin box.
- **Remote execution** This group is for executing workflows on a remote Spine engine. You can find instructions on how to set it up in *Spine Engine Server*

10.6 Application preferences

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

10.7 Where are the application settings stored?

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

CHAPTER

ELEVEN

WELCOME TO SPINE DATABASE EDITOR'S USER GUIDE!

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

11.1 Spine data structure

Documentation for the Spine data structure can be found here.

11.2 Getting started

This section gives a short outline on how to get started using the editor and how to navigate the ui. Information about the settings for the editor can be found in *Db editor Settings*.

11.2.1 Launching the editor

From Spine Toolbox

There are two different ways to open a single database in Spine database editor from Spine Toolbox:

Using a *Data Store* project item:

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

Without a *Data Store*:

1. From the main window select **File -> New DB Editor**.
2. Open the menu by clicking on the hamburger menu icon () or by pressing **ALT+F** or **ALT+E**.
3. Select **Open...** to open an existing database, **New..** to create a new one or paste a database URL into the URL bar.

From the command line

To open a single database in Spine database editor, use the `spine-db-editor` application which comes with Spine Toolbox. After the virtual environment is activated the editor can be opened with the following command:

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

Note that for e.g. an SQLite database, the url should start with `sqlite:///` followed by the path.

11.2.2 Adding multiple databases to one editor

It is possible to open multiple databases in the same editor. This allows one to view and modify the data of the open databases in one editor.

To open multiple SQLite databases in the same Spine database editor by file browser:

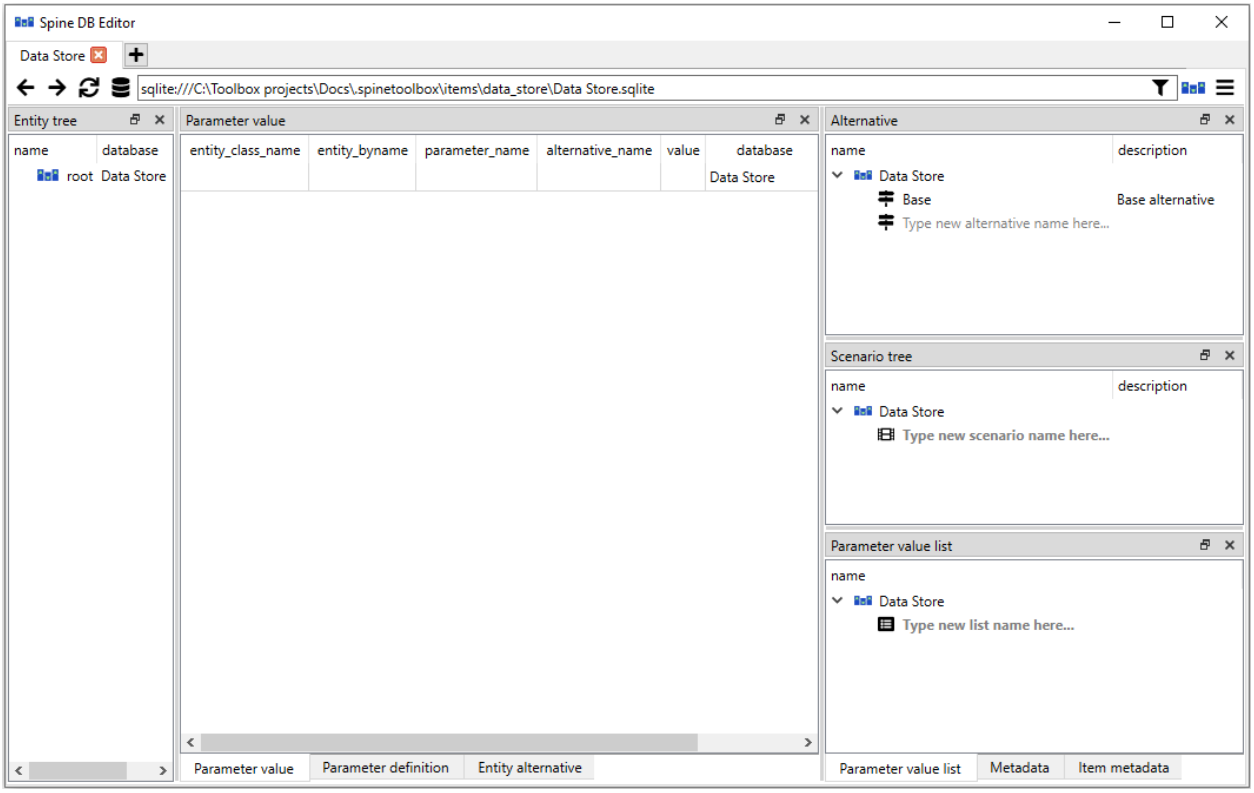
1. Open a database Database editor using any of the ways explained before.
2. Select **Add...** from the editor's hamburger menu ().
3. Browse to the directory of the SQLite file and open it.

By using the database URL:

1. Open a database Database editor using any of the ways explained before.
2. In the URL bar, after the already open database's URL add a semicolon `;` and after that the URL of the other database to be opened in the same editor.

11.2.3 Knowing the UI

When you open an empty database for the first time in a Spine Database Editor, it should look something like this:



The dock widgets can be scaled by dragging them from the sides and moved around by dragging them from their darker colored headers. Like with other widgets, Toolbox remembers the customizations and the editor will open in the same configuration when it is opened the next time. The dock configurations are URL specific. the configurations for the URL can be restored back to default from the hamburger menu **View->Docks...->Reset docks**.

Tab bar

The uppermost UI element is the tab bar. One editor window can have multiple tabs. New tabs can be added by pressing the plus-sign (**+**) in the tab bar. In the newly created tab, databases can be opened once again with the instructions given above. Tabs can be deleted from the editor by pressing the cross (**X**) inside a tab. The tabs can be dragged from the tab bar to create new editor windows. Tabs from different windows can also be dragged into others, fusing them into the same editor window.

Navigation bar

Right below the tab bar there is the navigation bar. With the backwards and forwards arrows it is possible to go back to the database that was previously loaded in the specific tab. This is kind of analogous of web browsers and going back to the previous page. Next to the arrows there is the **reload** () button. It can be used to reload the data of the database. Next up is the Data Store icon () which lists the Data Store items in the project and can be used to open any of them in the current tab. The URL bar contains the URL of the databases tha are currently open in the tab. As mentioned before, databases can be opened by inserting valid database URLs into this field and pressing enter. The URL bar also contains the filter (more about this later). After the URL bar there is the Spine-Toolbox logo which when clicked brings up the Spine-Toolbox main window. Finally there is the hamburger menu () which holds many of the functionalities of the Spine Database Editor.

Hamburger menu

The hamburger menu () can be located in the upper left corner of the Spine Database Editor. It is the place where options and other such things can be found.

File

The uppermost section in the menu is dedicated to actions related to databases. There you can create a new Spine db from **New...**, open an existing one from **Open...** or add another database to the current tab form **Add...** as explained before. There are also options **Import...**, **Export...** and **Export session...**. The importing works kind of like adding another database to the existing tab but instead of just opening the other database it brings all of the data from the other database and merges it into the current database. With export it is possible to export the current database into it's own `.sqlite` file. The export session works just like export but instead of exporting the whole database, it exports just the new modifications that have been made since the last commit.

Edit

In the **Edit** section there lies the **Undo** and **Redo** -buttons. These can be used to undo and redo the actions that have been made in the editor (**CTR+Z** and **CTR+Y** also work). The **Copy name(s) as text** allows the user to copy items into the clipboard that can then be pasted elsewhere. The **Paste** option does exactly what it says, it pastes the data on the clipboard into the selected field(s). The **Purge...** button is quite useful when there is a need to get rid of a lot of data quickly. Clicking it will open a new window where options for the purging are given. Find out more about purging in the section *Removing data*. The **Vacuum** option tries to free up some memory from the claws of the database.

View

The different view modes are listed in the **View** -section. Also the **Docks...** button for managing the visibility of the UI elements is located here. When switching to the **Value**, **Index** and **Element** views something need to selected from the entity tree in order for the view to show anything meaningful. The Graph view will show an graphical representation of the entities whereas the table view shows the plain data in table format. By pressing the **Docks...** one can customize what UI elements are displayed. This way it is possible to for example have the graph and scenario pivot table views open at the same time.

Session

The **Commit..** button is for committing the changes in the database. Pressing the button will open up a commit dialog box, where a commit message can be written. The default commit message is just "Updated" but it is good practise to write descriptive and concise messages. The **Rollback** button reverts the database to the state it was in when it was committed the last time. This means that all modifications to the data that haven't been committed will be lost. It is also good to note that this action clears the undo/redo stack which means that the operation is irreversible. The **History** button allows one to view the commit history of the database.

Other

In the bottom part of the hamburger menu there is a button to open the User Guide in a web browser, **Settings** button to open the Spine Database Editor settings and a **Close** button for closing the editor. More information about the settings can be found in *Db editor Settings*.

Filter

The filter can be used to select which items are allowed to be shown in the editor. The filter is based on scenarios. By pressing the filter image in the right end of the URL bar, the filter selector widget opens up. There the desired scenario can be selected. When a selection is made and the **Update filters** button is pressed, the changes will be applied to the editor. Now all entities, parameters etc. will be filtered out if they don't belong to the scenario specified in the filter.

Tip: Note that after applying the filter, the URL gets updated with some additional information about the filters. It is therefore possible to make changes to the filtering just by modifying the URL from the URL bar.

Undo and redo

Whenever changes are made to the data in the Spine Database Editor, the changes get stored into memory. This allows undoing and redoing the operations made in the editor. Buttons for these operations can be found in the hamburger menu and the usual shortcuts **Ctrl+Z** and **Ctrl+Y** work also. However if the changes are committed, the memory for the changes gets cleared meaning that the changes before the commit can't be undone anymore.

Views and trees

Spine Database Editor has the following main UI components:

- *Entity tree*: they present the structure of entities in all databases in the shape of a tree.
- *Table views* (*Parameter value*, *Parameter definition*, *Entity alternative*): they present entity data in the form of stacked tables.
- *Pivot table* and *Frozen table*: they present data in the form of a pivot table, optionally with frozen dimensions.
- *Graph view*: it presents the structure of classes and entities in the shape of a graph.
- *Parameter value list*: it presents parameter value lists available in the databases.
- *Alternative*: it presents alternatives defined in the databases in the shape of a tree.
- *Scenario tree*: it presents scenarios defined in the databases in the shape of a tree.
- *Metadata*: presents metadata defined in the databases.
- *Item metadata*: shows metadata associated with the currently selected entities or parameter values.

Tip: You can customize the UI from the **View** section in the hamburger menu. There the **Docks...** menu can be used to enable and disable the different UI components listed above.

Items from the trees can be selected by clicking them with the left mouse button and the views will react to the changes. By default, multiple items can be selected at the same time across the trees by holding down **Ctrl** while making the selections. This behavior can be flipped from the editor settings (**Ctrl+,**) by toggling the *Sticky selection* -setting.

In the next section you will learn more about the different UI components and views available in the editor

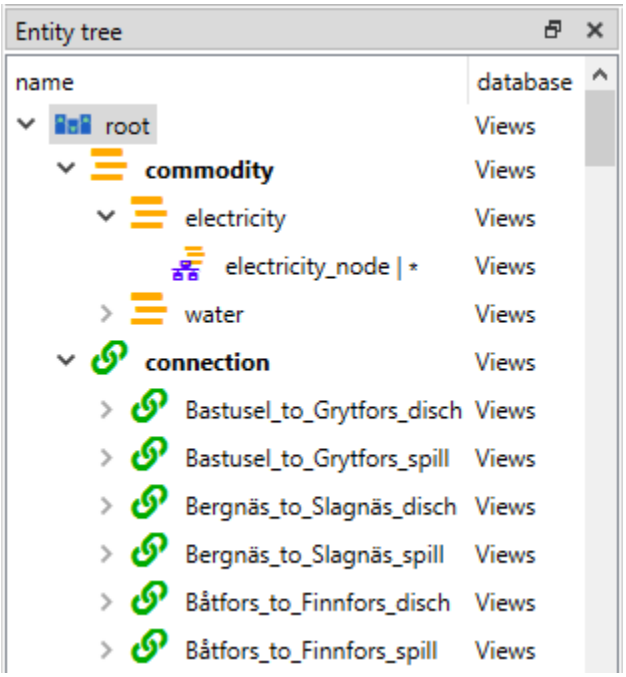
11.3 Viewing data

This section describes the available tools to view data.

11.3.1 Viewing entities and entity classes

Using the Entity Tree

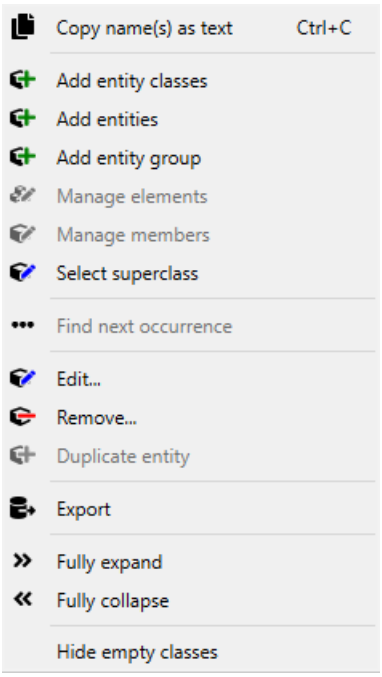
The **Entity Tree** presents the structure of entity classes and entities in all databases in the shape of a tree:



- To view all entity classes from all databases, expand the root item (automatically expanded when loading the form).
- To view all entities of a class, expand the corresponding entity class item.
- To view all multidimensional entities where a specific entity is an member, expand that entity.

Tip: To *extend* the selection in **Entity Tree**, press and hold the **Ctrl** key while clicking on the items.

Right clicking items in the **Entity Tree** will open up a context menu. Depending on what kind of item the menu was opened from (root, entity class, entity, N-D entity) all of the options might not be available. Unavailable options are still visible but they are greyed out:



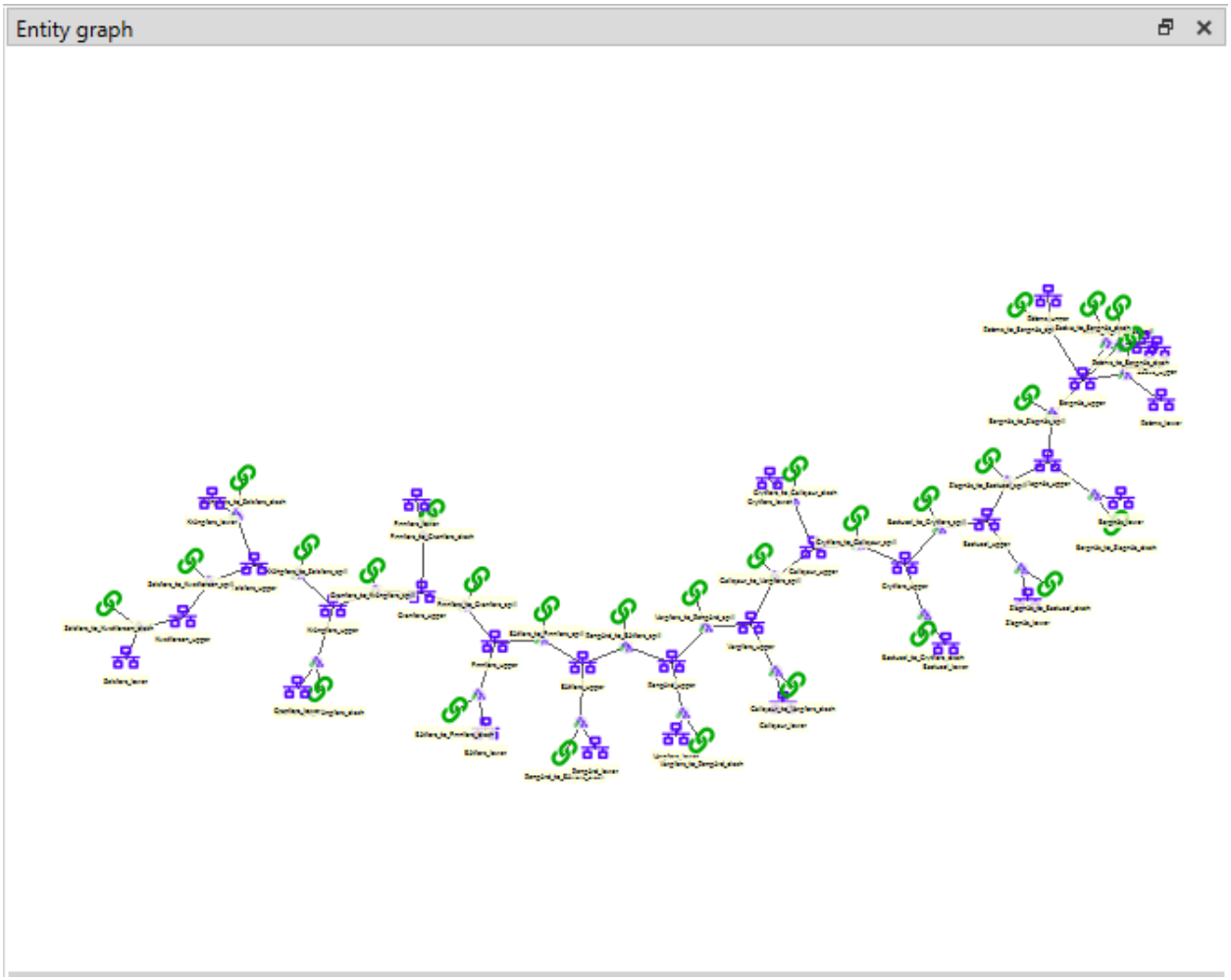
- **Copy name(s) as text** copies the data from the selection so that it can be pasted elsewhere.
- **Add entity classes** opens up a dialog to add new entity classes.
- **Add entities** opens up a dialog to add new entities.
- **Add entity group** opens up a dialog to create entity groups.
- **Manage elements** opens up a dialog where new entities can be created or existing ones deleted for a multidimensional entity.
- **Manage members** opens up a dialog to delete or add new members in an entity group.
- **Select superclass** opens up a dialog to set the superclass for a given entity class
- **Find next occurrence** goes to the next occurrence of the N-D entity in the **Entity Tree** and selects it. This can also be done by double-clicking the item.
- **Edit...** opens up a dialog where the name, description, icon and active by default -setting of an entity can be changed.
- **Remove...** removes the selection.
- **Duplicate entity** duplicates the whole entity.
- **Export** creates a new Spine Database in an *.sqlite* file with all of the relevant data to the selection.
- **Fully expand** expands the selection and all its children.
- **Fully collapse** collapses the selection and all its children.
- **Hide empty classes** whether to show empty classes in the tree or not.

Tip: To expand an item in **Entity Tree**, you can also double-click on the item or press the right arrow. This will only expand the next layer and leave the children expanded or collapsed depending on their previous state. Items in gray don't have any children, thus they cannot be expanded. To collapse an expanded item, double-click on it again or press the left arrow.

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

Using the Graph View

Graph View presents the structure of entities from one database in the shape of a graph:



Building the graph

To build the graph, select any number of items in the **Entity Tree**. What is included in the graph depends on the specific selection you make in the **Entity Tree**:

- To include all entities from the database, select the root item.
- To include all entities of an entity class, select the corresponding class item.
- To include specific entities, select them by holding down **Ctrl**.

Note: In **Graph View**, a small unnamed vertex represents a multidimensional entity with multiple elements, whereas

a bigger named vertex represents a zero dimensional entity. An arc between entities indicates that the 0-D entity is an element of that N-D entity.

The graph automatically includes N-D entities whenever *all* the elements of that entity are included (even if these entities are not selected in **Entity Tree**). You can change this behavior to automatically include N-D entities whenever *any* of the member elements are included. To do this, enable **Auto-expand entities** via the **Graph View**'s context menu.

Manipulating the graph

You can move items in the graph by dragging them with your mouse. By default, each items moves individually. Like in the **Design view**, multiple items can be moved at once by selecting them first.

To display **Graph View**'s context menu, just right-click on an empty space in the graph. The context menu has the following options:

- **Add entities…** opens up the add entities dialog, from where new entities can be added.
- **Search** highlights the specified entities with color so that they are easier to visualize.
- **Hide classes** can be used to disable all of the entities from an entity class from showing in the graph. **Show** can then be used to bring back the hidden classes one by one or **Show all** to bring them all back.
- **Prune classes** works like **Hide classes** but it also hides all the classes that have the specified class as an element. Once again these can be brought back one by one with **Restore** or all at once with **Restore all**.
- **Zoom** has three options: zoom out, zoom in and reset zoom. Using the scroll wheel of the mouse on the **Graph View** also works.
- **Arc-length** has two buttons: one for making the arcs between the entities longer and one for making them shorter.
- **Rotate** rotates the whole graph by 15° per step. Also can be done by holding down **SHIFT** while scrolling with the mouse wheel.
- **Auto-expand entities** If enabled, the graph will also include entities where the selections are members besides just the selections. if disabled, the graph will only show the selected entities.
- **Merge databases** Whether to merge the databases or not.
- **Snap entities to grid** makes it so that the placement of the entities can't be arbitrary anymore but instead they can only lay on a grid.
- **Max. entity dimension count** defines a cutoff for the number of dimensions an entity can have and still be drawn.
- **Number of build iterations** defines the maximum numbers of iterations the layout generation algorithm can make.
- **Minimum distance between nodes (%)** is used for setting the ideal distance between entities in the graph.
- **Decay rate of attraction with distance** The higher this number, the lesser the attraction between distant vertices when drawing the graph.
- **Select graph parameters** is where different aspects of the graph can be mapped to for example parameter values.
- **Select background image** can be used to set any *.svg* image as the background for the graph.
- **Save positions** Saves the positions of the items into the database. To clear the saved position select **Clear saved positions**.
- **Save state…** saves the drawn graph. Selecting a specific state from **Load state…** will load that state into the **Graph View**. Saved states can be deleted from **Remove state**.

- **Export as image...** can be used to export the image of the graph in either *.svg* or *.pdf* formats
- **Export as video...** can be used to export the video of the graph.
- **Rebuild** to rebuild the whole graph.

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

Note: Pruned items are remembered across graph builds.

To display an entity item's context menu, just right-click on it. The context menu has a few different options:

- To expand or collapse N-D entities, on an entities context menu hover **Expand** or **Collapse** and select the entity class from the popup menu.
- **Connect entities** allows the creation of new N-D entities straight from the **Graph View**. When hovering over the option, the list of relevant multi dimensional entity classes where the selected entity could possibly be a member are shown. After selecting one of the items in the list, the entities that you want to make up the new new entity in the selected entity class can be selected by clicking them in the graph. Once the selections are made, a popup showing the to be added entities is shown. By default every permutation of the selections is staged to be added but individual items can be also deselected.
- **Edit**, **Remove** and **Duplicate** work as they do in the **Entity Tree**.

11.3.2 Viewing parameter definitions and values as well as entity alternatives

Using Table Views

Table View's: *Parameter value*, *Parameter definition* and *Entity alternative* present entity data from all databases in the form of tables:

Parameter value

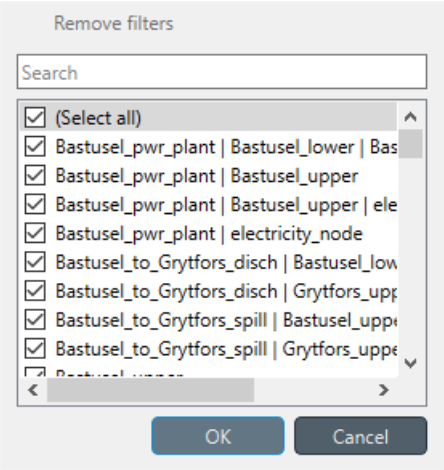
entity_class_name	entity_byname	parameter_name	alternative_name	value	database
model	instance	duration_unit	Base	hour	views
model	instance	model_end	Base	2019-01-...	views
model	instance	model_start	Base	2019-01-...	views
node	Bastusel_upper	demand	Base	-0.25797...	views
node	Bastusel_upper	fix_node_state	Base	Time series	views
node	Bastusel_upper	has_state	Base	True	views
node	Bastusel_upper	node_state_cap	Base	8208.0	views
node	Bastusel_upper	state_coeff	Base	1.0	views
node	Bergnäs_upper	demand	Base	-22.29	views
node	Bergnäs_upper	fix_node_state	Base	Time series	views
node	Bergnäs_upper	has_state	Base	True	views
node	Bergnäs_upper	node_state_cap	Base	216120.0	views
node	Bergnäs_upper	state_coeff	Base	1.0	views
node	Båtfors_upper	demand	Base	-2.0	views

To filter a **Table View** by any entities and/or classes, select the corresponding items in either **Entity Tree** or **Graph View**. To remove all these filters, select the root item in **Entity Tree**.

A **Table View** can also be filtered by selecting alternatives or scenarios from **Alternative** and **Scenario tree**. This filter is orthogonal to the entity/class filter and can be used together with it. To remove all these filters, simply select the root item in **Entity Tree** or deselect all items from **Alternative** and **Scenario tree**.

All the filters described above can be cleared with the *Clear all filters* item available in the right-click context menu of the **Table View**.

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



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

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

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

Entity alternative

Entity alternative provides a way to set which entities are active and which are not in each alternative:

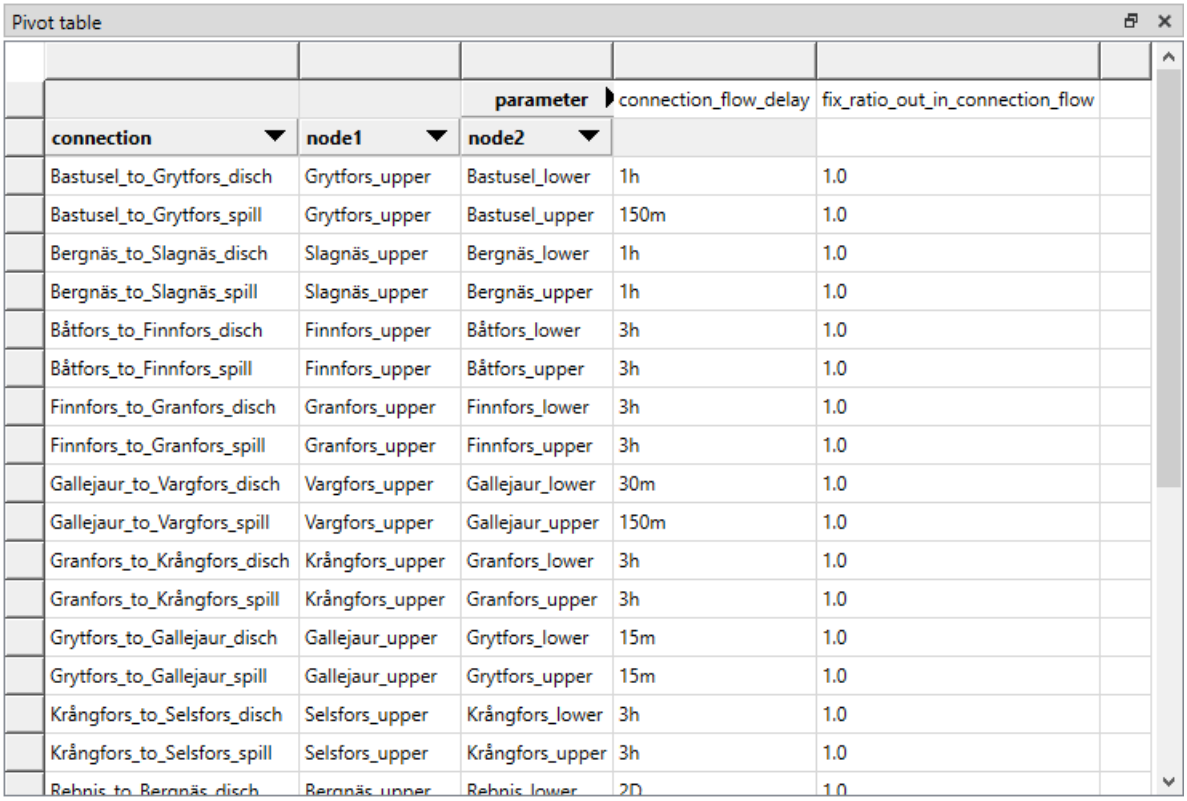
Entity alternative

entity_class_name	entity_byname	alternative_name	active	database
commodity	electricity	Base	true	views
commodity	water	Base	true	views
connection	Bastusel_to_Grytfors_disch	Base	false	views
connection	Bastusel_to_Grytfors_spill	Base	true	views
				views

11.3.3 Viewing parameter values and multidimensional entities

Using Pivot View and Frozen Table

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



		parameter	connection_flow_delay	fix_ratio_out_in_connection_flow
connection	node1	node2		
Bastusel_to_Grytfors_disch	Grytfors_upper	Bastusel_lower	1h	1.0
Bastusel_to_Grytfors_spill	Grytfors_upper	Bastusel_upper	150m	1.0
Bergnäs_to_Slagnäs_disch	Slagnäs_upper	Bergnäs_lower	1h	1.0
Bergnäs_to_Slagnäs_spill	Slagnäs_upper	Bergnäs_upper	1h	1.0
Båtfors_to_Finnfors_disch	Finnfors_upper	Båtfors_lower	3h	1.0
Båtfors_to_Finnfors_spill	Finnfors_upper	Båtfors_upper	3h	1.0
Finnfors_to_Granfors_disch	Granfors_upper	Finnfors_lower	3h	1.0
Finnfors_to_Granfors_spill	Granfors_upper	Finnfors_upper	3h	1.0
Gallejaur_to_Vargfors_disch	Vargfors_upper	Gallejaur_lower	30m	1.0
Gallejaur_to_Vargfors_spill	Vargfors_upper	Gallejaur_upper	150m	1.0
Granfors_to_Krångfors_disch	Krångfors_upper	Granfors_lower	3h	1.0
Granfors_to_Krångfors_spill	Krångfors_upper	Granfors_upper	3h	1.0
Grytfors_to_Gallejaur_disch	Gallejaur_upper	Grytfors_lower	15m	1.0
Grytfors_to_Gallejaur_spill	Gallejaur_upper	Grytfors_upper	15m	1.0
Krångfors_to_Selsfors_disch	Selsfors_upper	Krångfors_lower	3h	1.0
Krångfors_to_Selsfors_spill	Selsfors_upper	Krångfors_upper	3h	1.0
Rebnis_to_Bergnäs_disch	Bergnäs_upper	Rebnis_lower	2D	1.0

To populate the tables with data for a certain class, just select the corresponding class item in **Entity Tree**.

Selecting the input type

Pivot View and **Frozen Table** support four different input types:

- **Value** (the default): it shows entities, parameter definitions, alternatives, and databases in the headers, and corresponding parameter values in the table body.
- **Index**: Similar to the above, but it also shows parameter indexes in the headers. Indexes are extracted from special parameter values, such as time-series.
- **Element**: it shows entities, and databases in the headers, and corresponding multidimensional entities in the table body. It only works when a N-D entity is selected in the **Entity Tree**.
- **Scenario**: it shows scenarios, alternatives, and databases in the header, and corresponding *rank* in the table body.

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

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

Pivoting and freezing

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

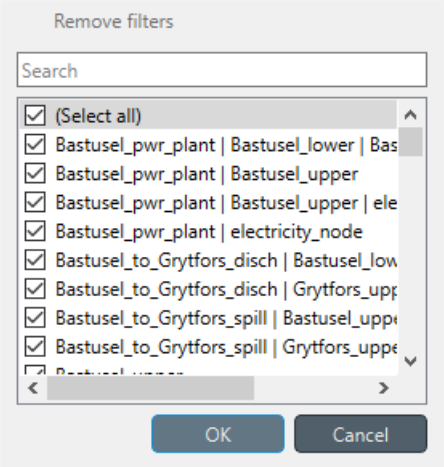
To freeze a dimension, drag the corresponding header block from **Pivot View** into **Frozen table**. To unfreeze a frozen dimension, just do the opposite.

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

Tip: If you are not seeing the data you think you should be seeing, it might be because there is some selection active in the **Frozen Table** that is filtering those values out of the **Pivot View**.

Filtering

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

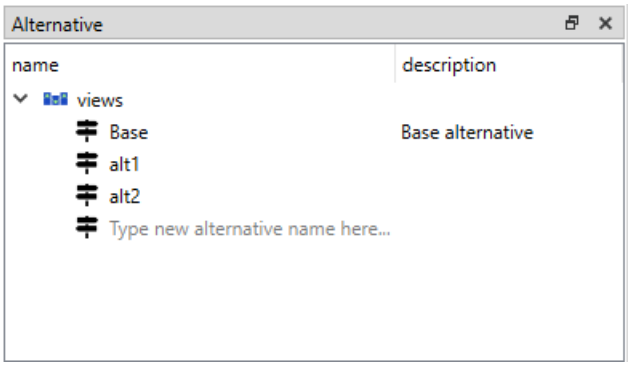


Uncheck the items you don't want to see in the table and press **Ok**. Additionally, you can type in the search bar at the top of the menu to filter the list of items. To remove the current filter, select **Remove filters**.

To filter the **Pivot View** by an individual vector across the frozen dimensions, select the corresponding row in **Frozen Table**.

11.3.4 Viewing alternatives and scenarios

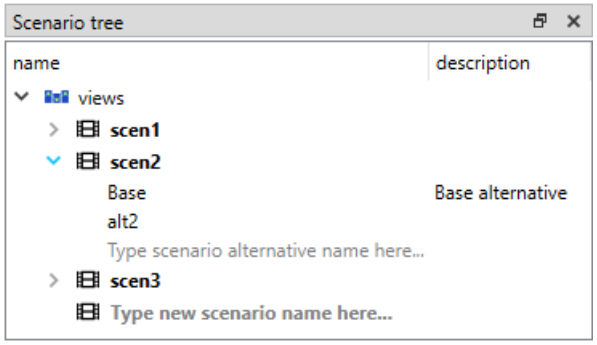
You can find alternatives from all databases under **Alternative**:



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

11.3.5 Viewing scenarios

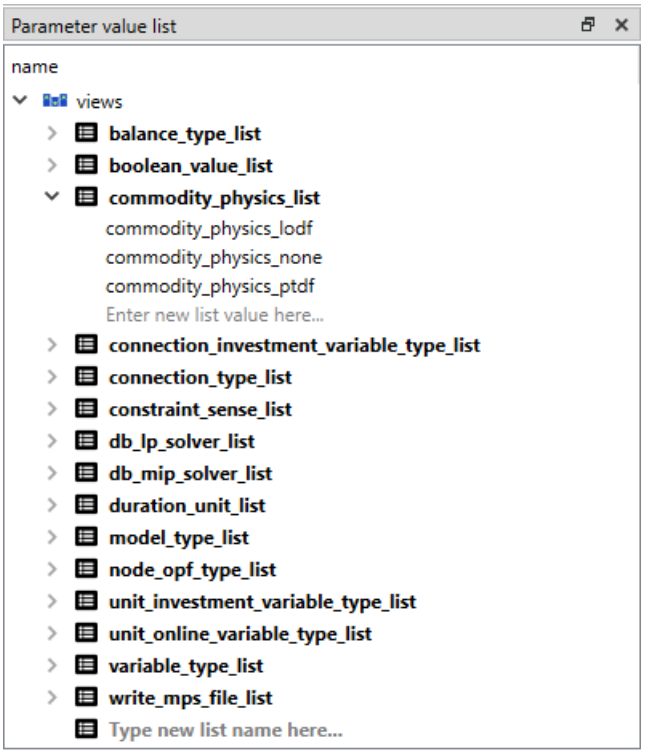
You can find scenarios from all databases under **Scenario tree**:



To view the scenarios from each database, expand the root item for that database. To view the alternatives for a particular scenario, expand the corresponding scenario item.

11.3.6 Viewing parameter value lists

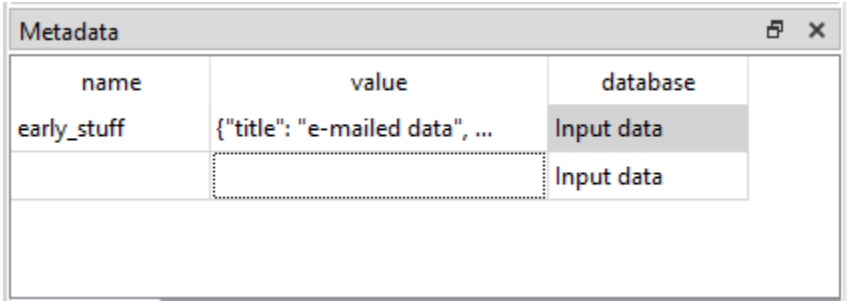
You can find parameter value lists from all databases under **Parameter value list**:



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

11.3.7 Viewing metadata

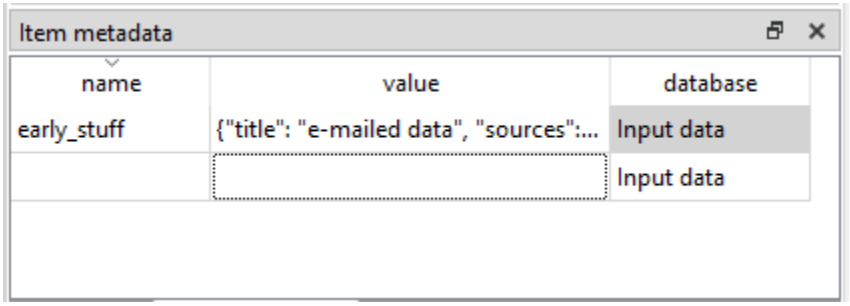
You can find metadata from all databases under **Metadata**:



See also *Spine Metadata Description*.

11.3.8 Viewing item metadata

You can find metadata for currently selected entities or parameter values under **Item metadata**:



11.4 Adding data

This section describes the available tools to add new data. Note that after adding data to a Spine Database, it still needs to be committed in order for the changes to take effect beyond just the current session in the Spine Database Editor. More information about this in the chapter *Committing and History*.

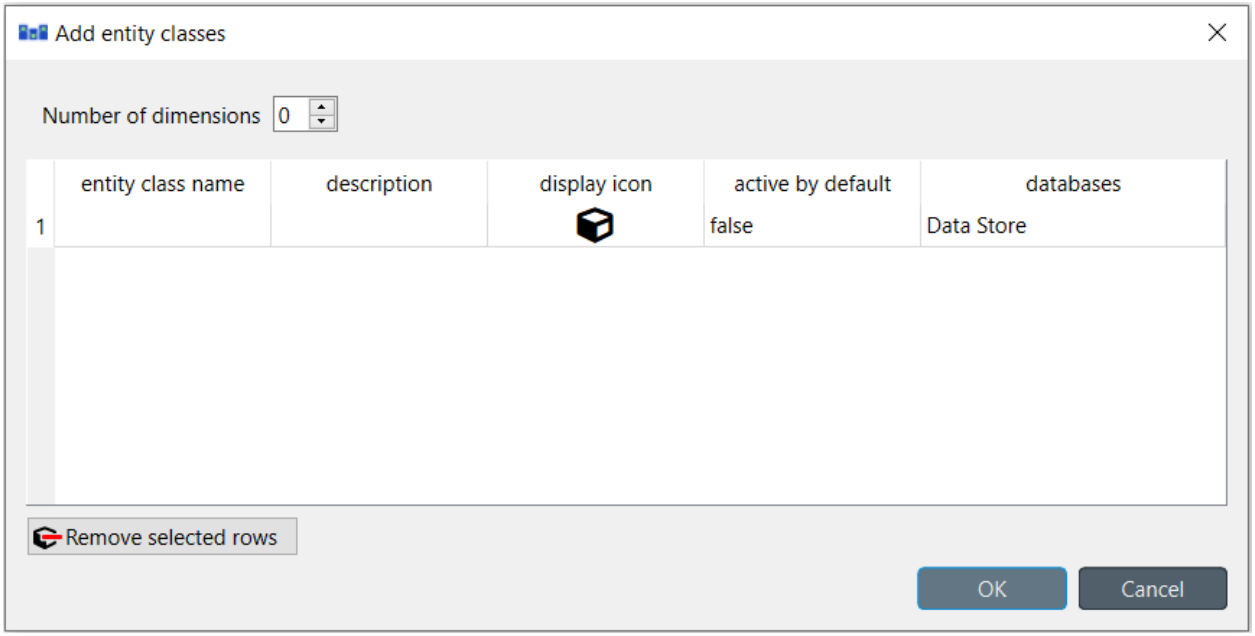
- *Adding parameter value lists*
- *Adding metadata and item metadata*

11.4.1 Adding entity classes

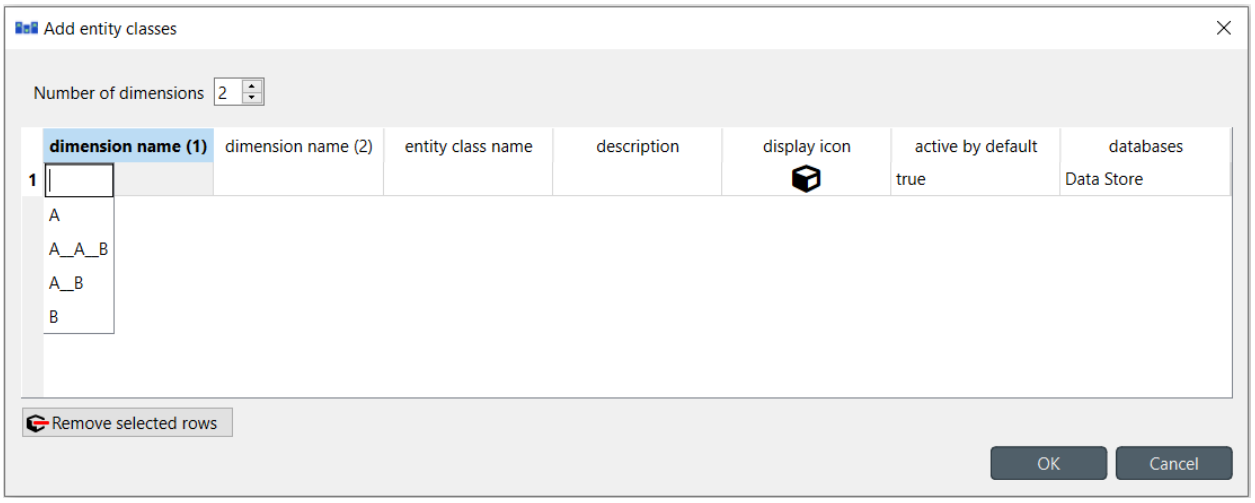
From Entity Tree

Right-click on the root item in **Entity Tree** to display the context menu, and select **Add entity classes**.

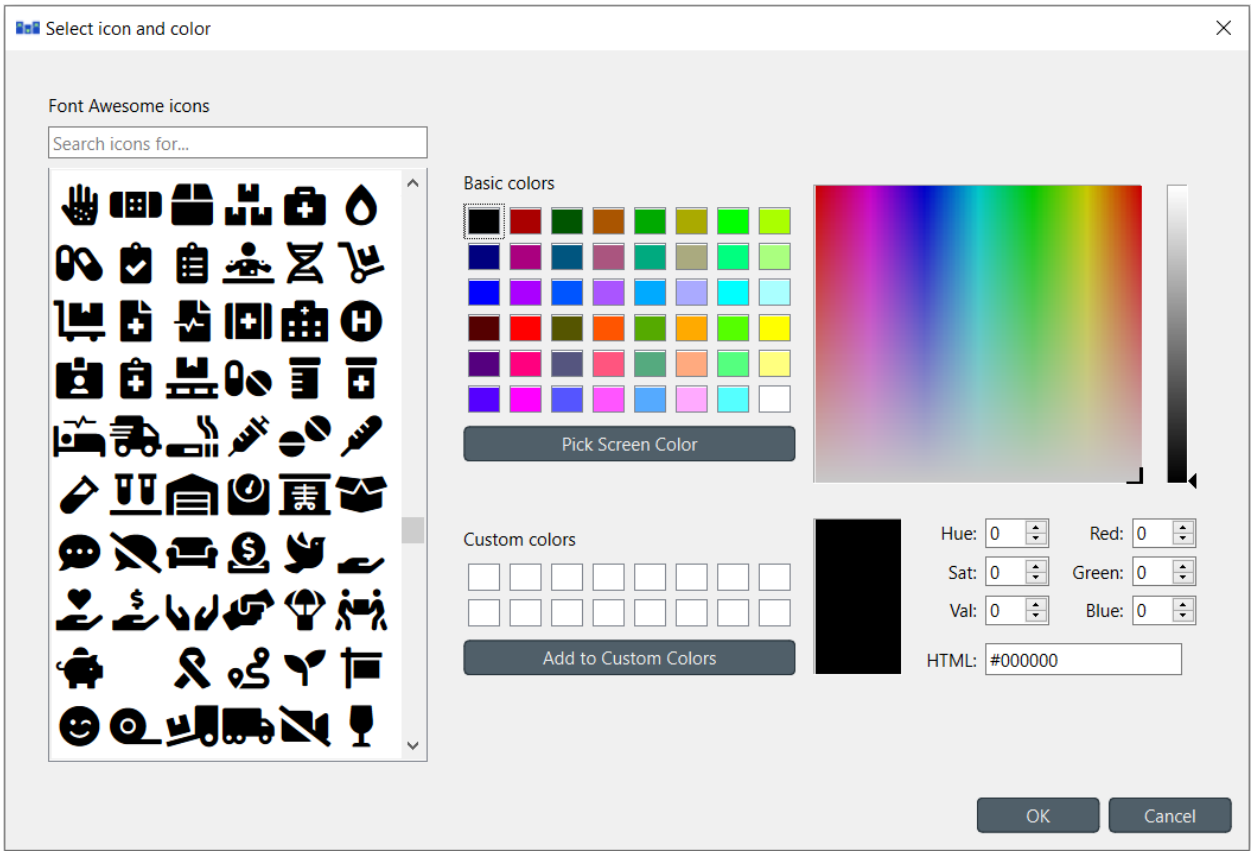
The *Add entity classes* dialog will pop up:



Select the number of dimensions using the spinbox at the top. The amount of dimensions determines the number of dimension names that need to be selected. With 0-dimensional classes, like in the image above, only the name of the created entity class is required. The class's name is to be entered below the header **entity class name**. In other cases like in the image below, in addition to the created classes name, also the classes making up the new class need to be selected:



They need to be filled under the headers **dimension name (1)** through **dimension name (N)** where N is the selected dimension. Note that because of this, there needs to be at least N entity classes already defined in the database when creating an N dimensional entity class. To display a list of available classes, start typing or double click on the cell below the header. Optionally, you can enter a description for each class under the **description** header. Double clicking the cell under the header **display icon** will open up the icon editor where the visual representation of the class can be modified:



The boolean value of **active by default** will determine whether the entities created under the created class will have the value under entity alternative set as true or false by default. Finally, select the databases where you want to add the entity classes under **databases**.

Multiple additions can be at once. When some information is inserted into the preceding row, a new empty row will appear in the dialog where a new class with the same dimensions can be defined. Delete entire rows from the dialog with **Remove selected rows**. When you're ready, press **Ok** to make the additions.

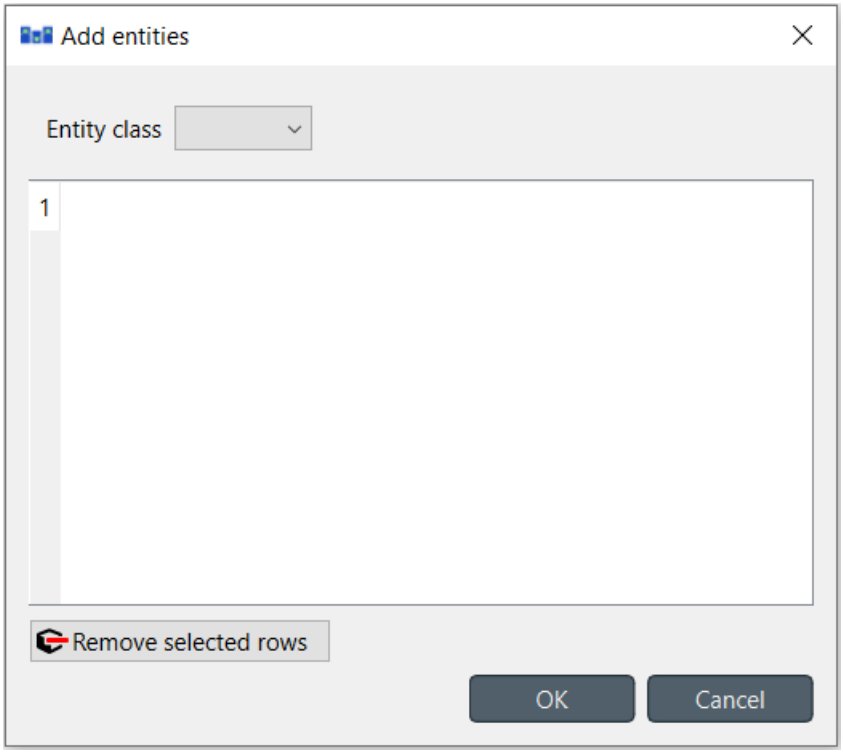
Tip: All the *Add...* dialogs support pasting tabular (spreadsheet) data from the clipboard. Just select any cell in the table and press **Ctrl+V**. If needed, the table will grow to accommodate the exceeding data. To paste data on multiple cells, select all the cells you want to paste on and press **Ctrl+V**.

11.4.2 Adding entities

From Entity Tree or Graph View

Right-click on the root item in **Entity Tree** and select **Add entities**, or click on an empty space in the **Graph View** and select **Add entities...** from the context menu.

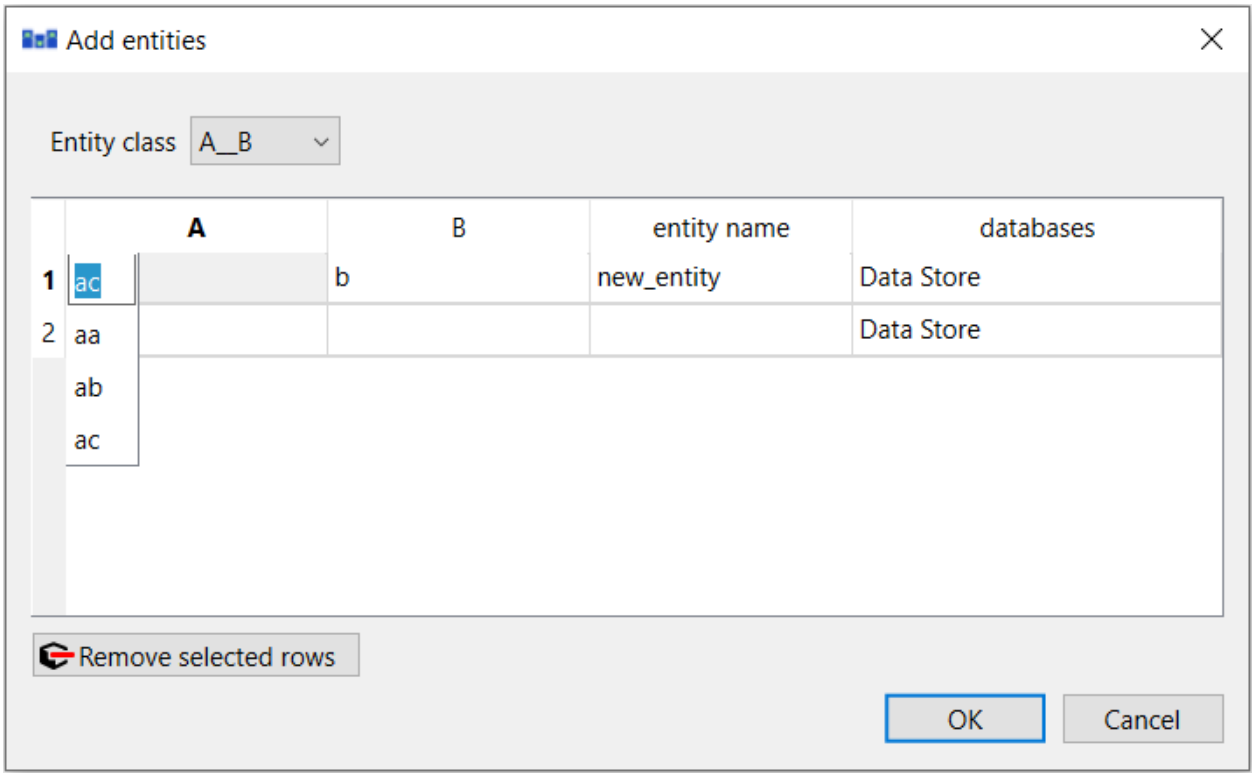
This will open up the **Add entities** dialog:



Select the class where you want to add the entities from **Entity class**. It will list all of the entity classes. To narrow down the list, instead of opening the dialog from the root item, open it from a specific entity class item in the **Entity Tree**. This way the class will be preselected from the list and the list will overall only contain other classes that are relevant to the selected class.

Enter the names of the entities under **entity name**. Finally, select the databases where you want to add the entities under **databases**. When you're ready, press **Ok**. Rows can once again be deleted with the **Remove selected rows** -button.

With N-D entity classes, the elements need to be specified. After defining the elements the entity's name can be modified:



New entities for an existing N-D entity class can also be created easily from the **Graph view**. Make sure all the entities you want as members in the new entity are in the graph. To start the new N-D entity, either double click on one of the entity items in the graph, or right click on it to display the context menu, and choose the class from **Connect entities**. After selecting the class the mouse cursor will adopt a cross-hairs shape.

When hovering over a entity item, the cursor will aid by indicating an entity that can't be a member by turning into a red restriction -sign. When nearly all of the selections made and only the last member needs to be selected, the cursor will turn into a green checkmark when hovering over an appropriate entity. Click on each of the remaining member entities one by one to add them to the new entity. Once you've added enough members for the entity class, a dialog will pop up. In the dialog, all of the possible permutations of the selected members are presented. Check the boxes next to the entities you want to add, and press **OK**.

From Pivot View

To add an entity to a specific 0-D entity class, bring the class to **Pivot View** using either **Value** or **Index** (see *Using Pivot View and Frozen Table*). There under the class name just type a new name and the new entity will be added under the class. Note that is only possible to add 0-D entities this way even if you have selected an N-D class from the **Entity Tree**.

To enter a new entity to an N-D class, select the **Element** -view from the hamburger menu. This view contains all of the possible combinations of elements in the selected class. The entities can be added by checking the boxes and removed by unchecking them.

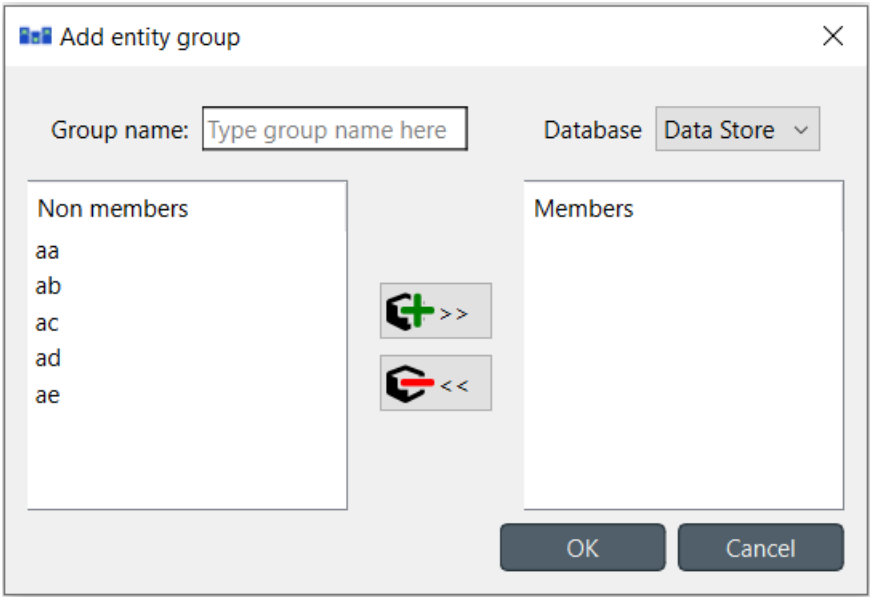
Duplicating entities

To duplicate an existing entity with all its parameter values and other associated data, right-click over the corresponding entity item in **Entity Tree** to display the context menu, and select **Duplicate entity**. The new entity will have the same name with an added (1) to indicate that it is a copy of the original entity. It can be renamed to be something else afterwards.

11.4.3 Adding entity groups

Right-click on an entity class item in **Entity Tree**, and select **Add entity group** from the context menu.

The **Add entity group** dialog will pop up:



Enter the name of the group, and select the database where you want the group to be created. Select the members under *Non members*, and press (>>) to add the members and (<<) to remove them. Multiple selection is supported with **Ctrl** and **Shift**. Finally press **OK** to create the group.

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

11.4.4 Adding parameter definitions

From Table View

To add new parameter definitions for an entity class, just fill the last empty row of *Parameter definition*. Only two of the fields are required when creating a new parameter definition: *entity_class_name* and *parameter_name*. Enter the name of the class under *entity_class_name*. To display a list of available entity classes, start typing in the empty cell or double click it. For the name of the parameter choose something that isn't already defined for the specified entity class. Optionally, you can also specify a parameter value list, a default value and a description.

In the column *value_list_name* a name for a parameter value list can be selected. Leaving this field empty means that later on when creating parameter values with this definition, the values are arbitrary. Meaning that the value could for example be a string or an integer. When the parameter value list is defined in the parameter definition, only the values in the list will be allowed to be chosen. For the creation of parameter value lists, see *Adding parameter value lists*.

In the *default_value* field, the default value can be set. The default value can be used in cases where the value is not specified. The usage of *default_value* is really tool dependent, meaning that the Spine Database Editor doesn't use the information of the default value anywhere, but it is instead left to the tool creators on how to utilize the default value. A short description for the parameter can be written in the *description* column.

The parameter is added when the background of the cells under *entity_class_name* and *database* become gray.

From Pivot View

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

11.4.5 Adding parameter values

From *Table View*

To add new parameter values for an entity, just fill the last empty row of the *Parameter value* -table. Enter the name of the class under *entity_class_name*, the name of the entity under *entity_byname*, the name of the parameter under *parameter_name*, and the name of the alternative under *alternative_name*. Optionally, you can also specify the parameter value right away under the *value* column. The database where the value will be added to is displayed in the last column of the table. To display a list of available entity classes, entities, parameters, or alternatives, just start typing or double click under the appropriate column. The parameter value is added when the background of the cells under *entity_class_name* and *database* become gray.

Note: To add parameter values for a 0-D entity, the entity has to exist beforehand. However, when adding parameter values for an N-D entity, you can specify any valid combination of elements by double clicking the cell under *entity_byname*, which opens up the *Select elements* -dialog. The specified N-D entity will be created if it doesn't yet exist.

From Pivot View

To add parameter value for any entity, bring the corresponding class to **Pivot View** using the **Value** input type (see *Using Pivot View and Frozen Table*). Then, enter the parameter value in the corresponding cell in the table body.

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

11.4.6 Adding entity alternatives

To add an entity alternative, open the **Entity Alternative -Table View**. There under *entity_class_name* select the class. Under *entity_byname* select the specific entity from that class and from *alternative_name* select the alternative. Then set the value of the *active* -column to either true or false by double clicking it. The background of the cells under *entity_class_name* and *database* should become gray, indicating that the entity alternative has been added.

11.4.7 Adding alternatives

From Alternative

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

You can also copy and paste alternatives between different databases.

From Pivot View

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

11.4.8 Adding scenarios

From Scenario Tree

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

To add an alternative for a particular scenario, drag the alternative item from **Alternative** and drop it under the corresponding scenario. The position where you drop it determines the alternative's *rank* within the scenario. Alternatives can also be copied from **Alternative** and pasted at the appropriate position in **Scenario Tree**.

If it is desirable to base a scenario on an existing one, scenarios can be duplicated using the **Duplicate** item in the right-click context menu. It is also possible to copy and paste scenarios between databases.

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

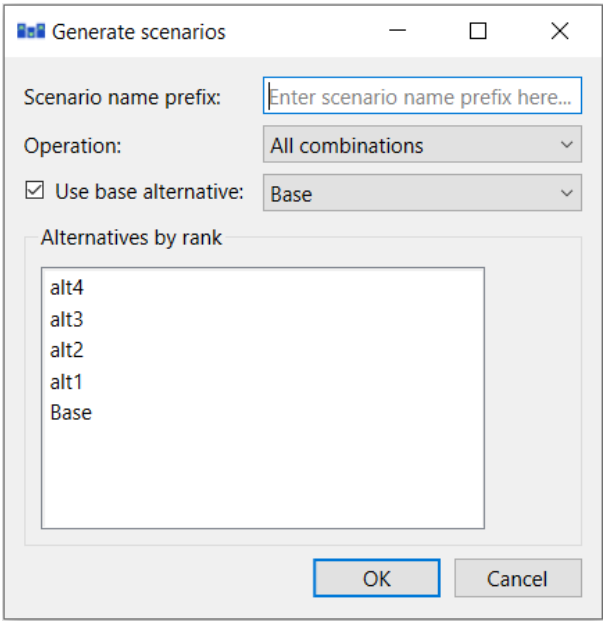
Note: As noted in the tooltip, scenario names longer than 20 characters may become shortened in generated files. This can happen for example when exporting the scenarios using the Exporter -project item. This can lead to confusion later on if the first 20 characters of the scenario names are identical. Therefore it is recommended to have a unique identifier for each scenario in the first 20 characters of its name.

From Pivot View

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

From Generate scenarios

Scenarios can be added also by automatically generating them from existing alternatives. Select the alternatives in **Alternative** (using **Ctrl** and **Shift** while clicking the items), then right click to open a context menu. Select **Generate scenarios...**



Give the scenario names a prefix. An index will be appended to the prefix automatically: **prefix01**, **prefix02**,... Select appropriate operation from the **Operation** combo box. Checking the **Use base alternative** check box will add the selected alternative to all generated scenarios as the lowest rank alternative. The **Alternative by rank** list allows reordering the ranks of the alternatives.

11.4.9 Adding parameter value lists

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

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

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

11.4.10 Adding metadata and item metadata

To add new metadata go to **Metadata** and add a new name and value to the last row.

To add a new link metadata for an item, select an entity from one of the entity trees or a parameter value from one of the parameter value tables. Then go to **Item metadata** and select the appropriate metadata name and value on the last row.

11.5 Updating data

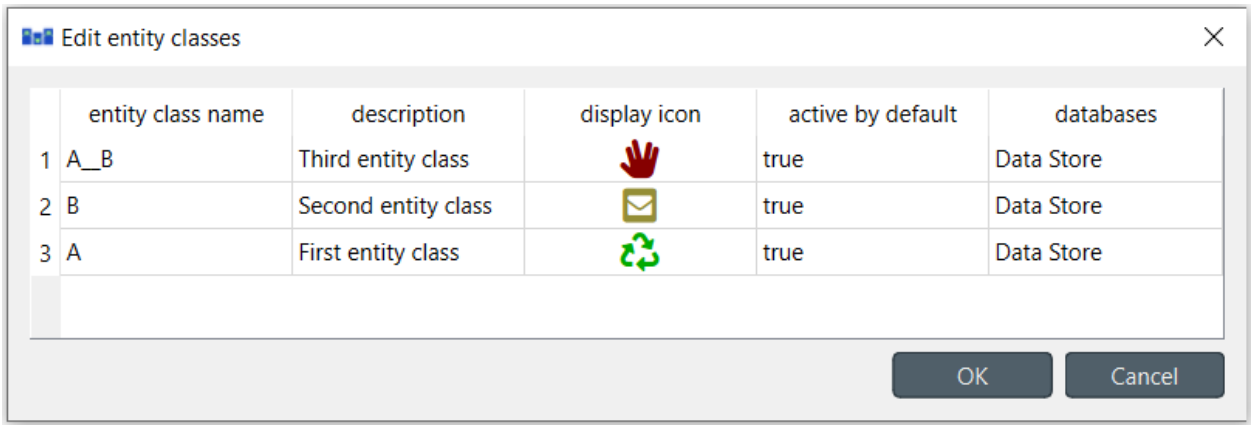
This section describes the available tools to update existing data.

11.5.1 Updating entities and classes

From Entity Tree or Graph View

Select any number of entity and/or class items in **Entity Tree**, or any number of entity items in **Graph View**. Then, right-click on the selection and choose **Edit…** from the context menu.

Depending on the selections, at least one *Edit…* dialog will pop up, and the tables will be filled with the current data of selected items. E.g.:



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

From Pivot View

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

11.5.2 Updating parameter definitions and values

From Table Views

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

From Pivot View

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

To modify parameter values for an entity, bring the corresponding class to **Pivot View** using the **Value** input type (see *Using Pivot View and Frozen Table*). Then, just edit the appropriate cell in the table body.

11.5.3 Updating entity alternatives

To update an entity alternative, edit the corresponding row from **Entity Alternative** in **Table View**.

11.5.4 Updating alternatives

From Pivot View

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

From Alternative

To rename an alternative, just edit the appropriate item in **Alternative**.

11.5.5 Updating scenarios

From Pivot View

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

To change the alternatives of a scenario as well as their ranks, check or uncheck the boxes on the pivot table. The number in the checkbox signifies the alternative's rank.

From Scenario Tree

To rename a scenario, just edit the appropriate item in **Scenario Tree**.

To change scenario alternative ranks, just drag and drop the items under the corresponding scenario.

11.5.6 Updating parameter value lists

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

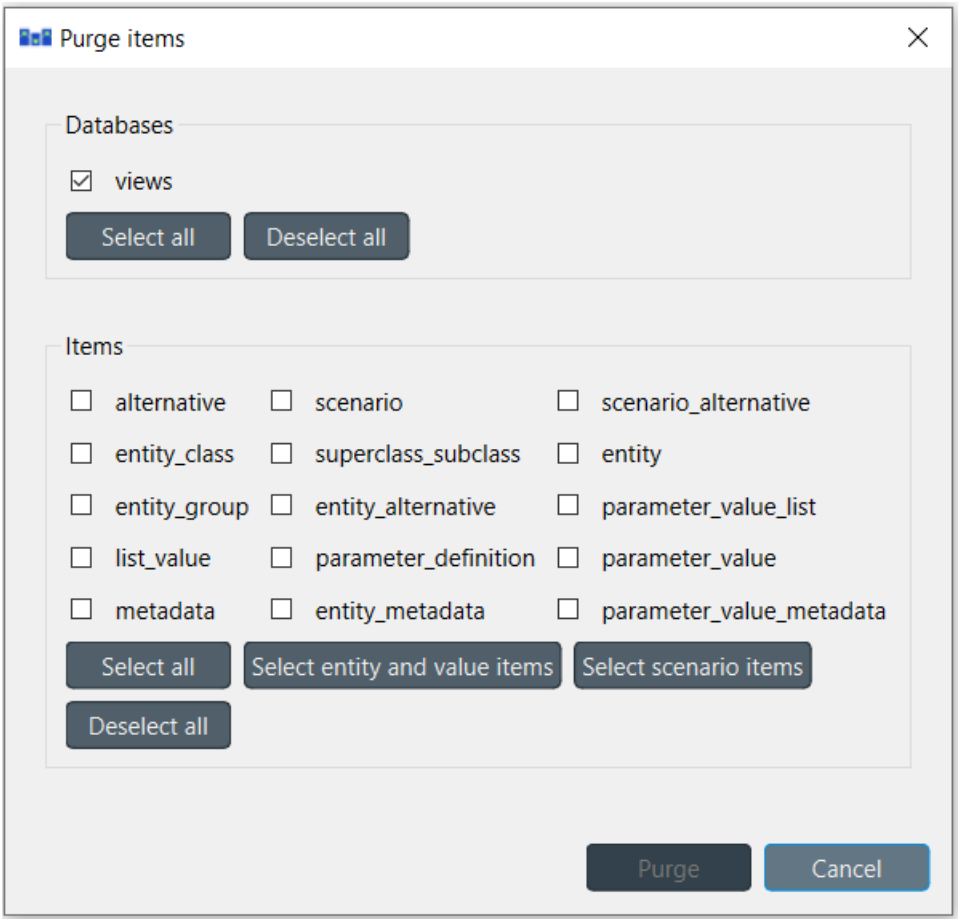
11.6 Removing data

This section describes the available tools to remove data.

- *Removing metadata*
- *Removing item metadata*

11.6.1 Purging items

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



The databases that are opened in the Editor are listed under *Databases*. From there you can select the databases where the mass removal will take place. The *Select all* -option will check all of the boxes and *Deselect all* will in turn uncheck every box.

The type of items that are to be deleted, need to be specified under *Items*. There are a couple of useful buttons in addition to the same *Select all* and *Deselect all*: *Select entity and value items* and *Select scenario items*. The former will select the *entity*, *entity_group*, *parameter_value*, *entity_metadata* and *parameter_value_metadata* items in the list. The latter will select the *alternative*, *scenario* and *scenario_alternative* items. When you are happy with your choices, press **Purge** to perform the mass removal.

Note: The purge dialog can also be opened from the **Properties** -dock widget of a Data Store.

Tip: Purging can also be an automated part of the workflow. See *Links* for more information about purging a database

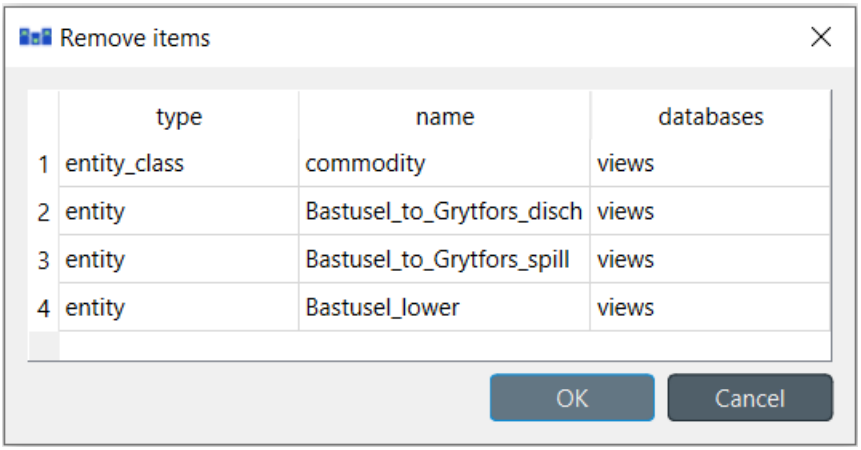
automatically.

11.6.2 Removing entities and classes

From Entity Tree or Graph View

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

The *Remove items* dialog will popup:



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

From Pivot View

To remove entities from a specific class, bring the class to **Pivot View** using the **Value** input type (see *Using Pivot View and Frozen Table*), and select the cells in the table headers corresponding to the entities you want to remove. Then, right-click on the selection and choose the **Remove entities** option from the context menu. This will remove the selected rows.

Alternatively, to remove N-D entities of a specific class, bring the class to **Pivot View** using the **Element** input type (see *Using Pivot View and Frozen Table*). The **Pivot View** headers will be populated with all possible combinations of entities across the member classes. Locate the member entities you want to remove, and uncheck the corresponding box in the table body.

11.6.3 Removing parameter definitions and values

From Table View

To remove parameter definitions or values, go to the relevant **Table View** and select any cell in the row corresponding to the items you want to remove. Then, right-click on the selection and select the **Remove row(s)** option from the context menu. Multiple selection is supported and the removal can also be performed by pressing **Ctrl+Del**.

From Pivot View

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

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

11.6.4 Removing alternatives

From Pivot View

Select the **Scenario** input type (see *Using Pivot View and Frozen Table*). To remove alternatives, select the to be removed items in the **alternative** header, right-click on the selection and choose **Remove alternatives** from the context menu.

From Alternative

To remove an alternative, just select the corresponding items in **Alternative**, right-click on the selection and choose **Remove** from the context menu.

11.6.5 Removing scenarios

From Pivot View

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

From Scenario Tree

To remove a scenario, just select the corresponding items in **Scenario Tree**, right-click on the selection and choose **Remove** from the context menu.

To remove a scenario alternative from a scenario, select the corresponding alternative items in **Scenario Tree**, right-click on the selection and choose **Remove** from the context menu.

11.6.6 Removing parameter value lists

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

11.6.7 Removing metadata

Select the corresponding items in **Metadata**, right-click on the selection and choose **Remove row(s)** from the context menu.

11.6.8 Removing item metadata

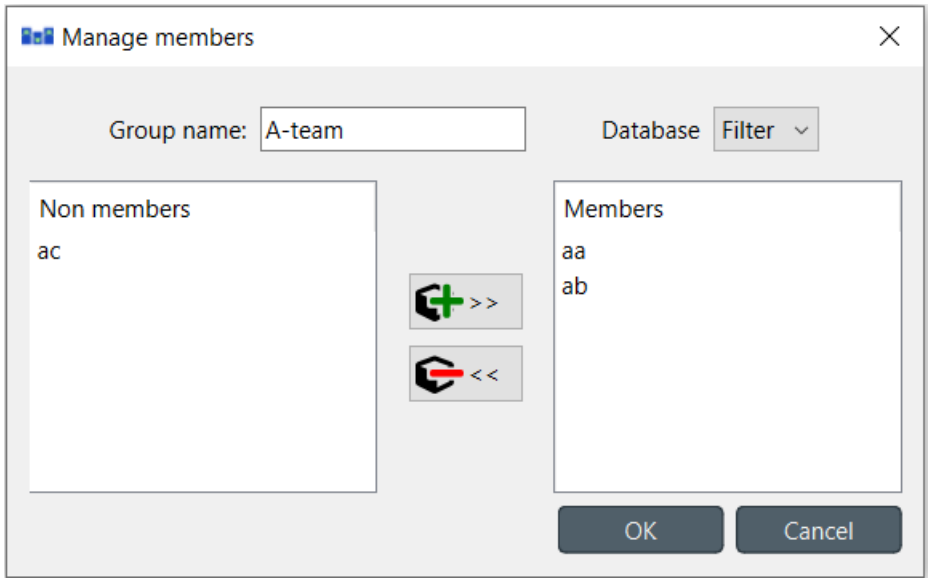
Select the corresponding items in **Item metadata**, right-click on the selection and choose **Remove row(s)** from the context menu.

11.7 Managing data

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

11.7.1 Managing entity groups

To modify entity groups, expand the corresponding entity class item in **Entity Tree** to display the group item, right-click on it and select **Manage members** from the context menu. The *Manage members* dialog will pop up:



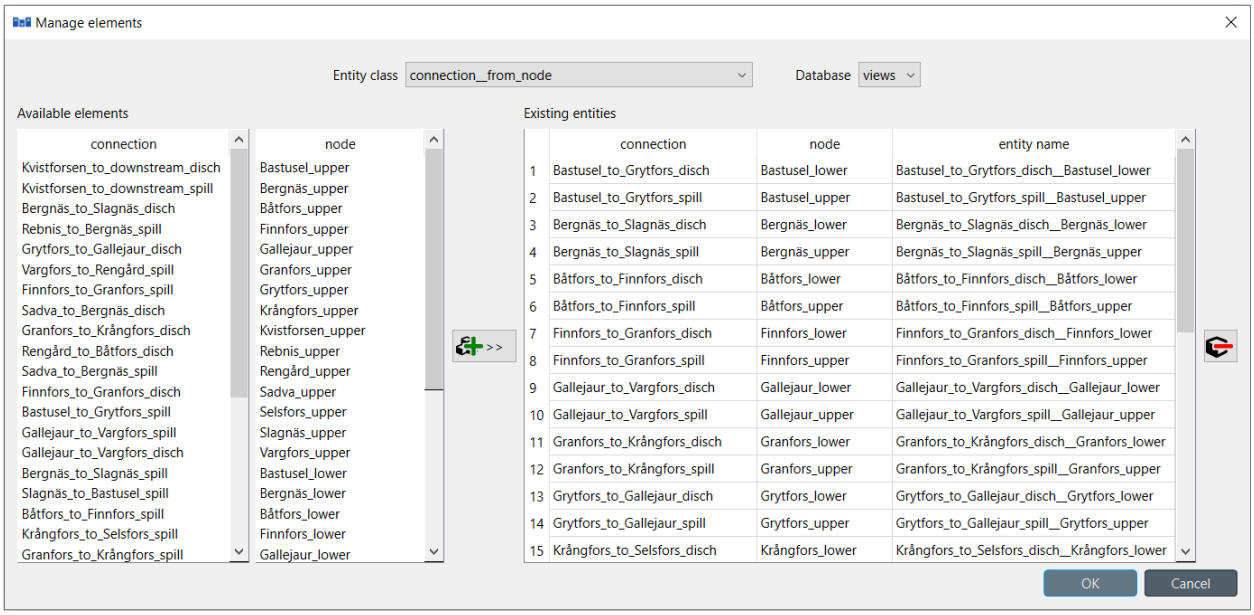
To add new member entities, select them under *Non members*, and press the (>>) button in the middle. To remove current members, select them under *Members*, and press the (<<) button in the middle. Multiple selection works in both lists.

When you're happy with the members, press **OK**.

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

11.7.2 Managing N-D entities

Right click the root item, or an N-D entity item in **Entity Tree** and from the context menu select **Manage elements**. The *Manage elements* dialog will pop up:



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

To add entities, select the elements for each class under *Available elements* and press the add button (>>) in the middle of the form. The entities will appear at the top of the table under *Existing entities*.

To add multiple entities at the same time, select multiple elements for one or more of the classes. All possible permutations of the selected elements will be added to *Existing entities*.

Tip: To *extend* the selection of entities for a class, press and hold the **Ctrl** key while clicking on more items. Holding down **Shift** allows to select an area of items by clicking the start and end of the selection.

To remove entities, select the appropriate rows under *Existing entities* and press the remove button () on the right.

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

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

11.8 Importing and exporting data

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

11.8.1 Overview

Spine Database Editor supports importing and exporting data in three different formats: SQLite, JSON, and Excel. The SQLite import/export uses the Spine Database format. The JSON and Excel import/export use a specific format described in *Format specifications*.

11.8.2 Importing

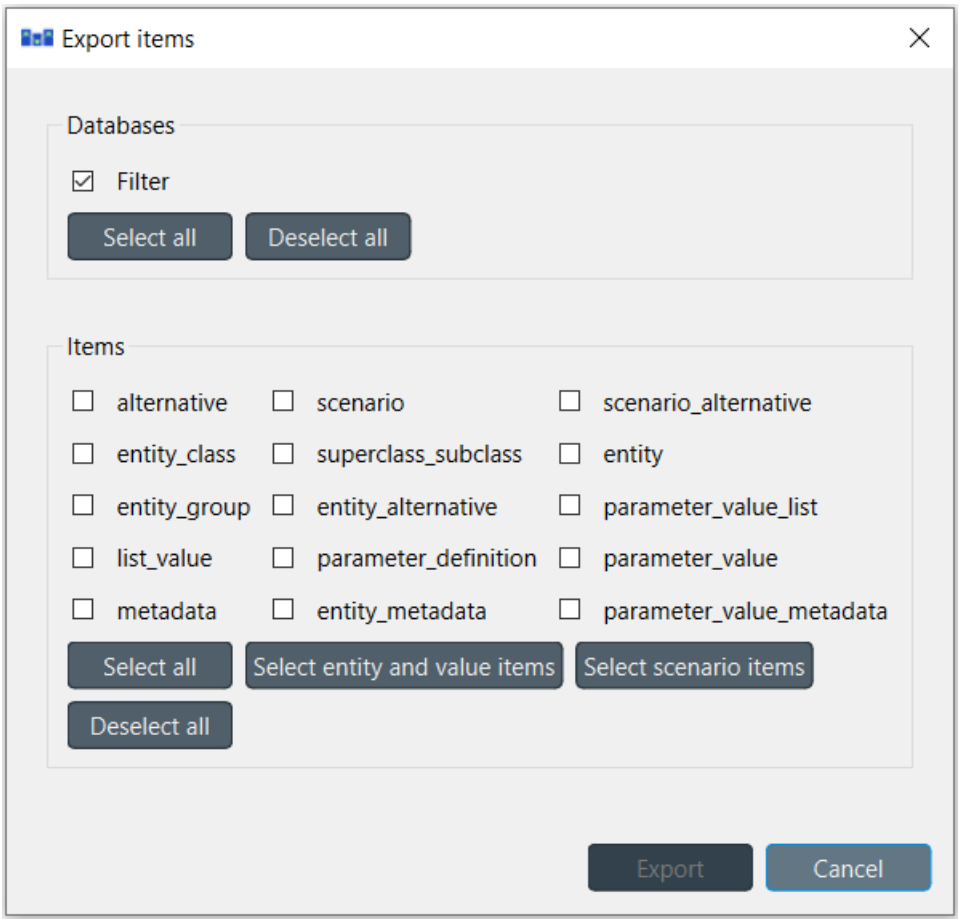
To import a file, select **File –> Import** from the hamburger menu. The *Import file* dialog will pop up. Select the file type (SQLite, JSON, or Excel), enter the path of the file to import, and accept the dialog.

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

11.8.3 Exporting

Mass export

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



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

Selective export

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

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

Session export

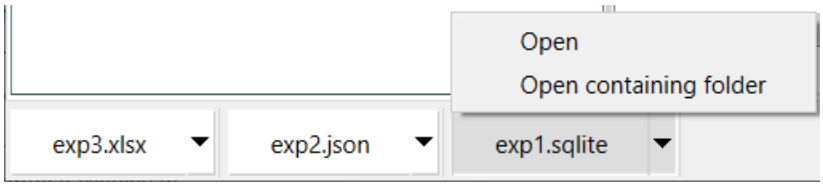
To export only uncommitted changes made in the current session, select **File –> Export session** from the hamburger menu.

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

Note: Export operations include all uncommitted changes.

11.8.4 Accessing/using exported files

Whenever you successfully export a file, a button with the file name is created in the *Exports* bar at the bottom of the form. Pressing that button will open the JSON or Excel file with the default program that your system associates with that filetype. Exports of SQLite file type will be opened in a new tab of the Spine Database Editor. To open the folder containing the export, click on the arrow next to the file name and select **Open containing folder** from the popup menu.



11.8.5 Format specifications

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

Excel format

Note: Excel exports are not comprehensive. Even though every type of item is selectable in the exporting selection, sheets will be generated only for some of the selections. Things like metadata and parameter value lists don't currently have export support with excel. The JSON export on the other hand is comprehensive and will export every detail about the database.

When parameter values are exported, the generated Excel will have every entity class on its own sheet. If the entity has indexed values (time-series, map etc.) as well as single values (floats, strings etc.) the entity will have more than one sheet, one containing the single values and others that unpack the indexed values:

scalar parameter data:

	A	B	C	D	E	F
1	sheet_type	entity				
2	class_name	node				
3	entity_dim_count	0				
4	value_type	single_value				
5	index_dim_count	0				
6						
7	node	alternative	demand	has_state	node_state_cap	state_coeff
8	Bastusel_upper	Base	-0,2579769	TRUE	8208	1
9	Bergnäs_upper	Base	-22,29	TRUE	216120	1
10	Båtfors_upper	Base	-2	TRUE	1330	1
11	Finnfors_upper	Base	0	TRUE	300	1
12	Gallejaur_upper	Base	-15,356963	TRUE	3600	1
13	Granfors_upper	Base	0	TRUE	280	1
14	Grytfors_upper	Base	-3,78	TRUE	1248	1
15	Krångfors_upper	Base	0	TRUE	330	1
16	Kvistforsen_upper	Base	-1,327381	TRUE	1120	1
17	Rebnis_upper	Base	-3,68	TRUE	205560	1
18	Rengård_upper	Base	-10,37	TRUE	1400	1
19	Sadva_upper	Base	-5,43	TRUE	168000	1
20	Selsfors_upper	Base	0	TRUE	500	1
21	Slagnäs_upper	Base	0	TRUE	768	1
22	Vargfors_upper	Base	-3,5584954	TRUE	4008	1

indexed parameter data:

	A	B	C	D	E	F
1	sheet_type	entity				
2	class_name	node				
3	entity_dim_count	0				
4	value_type	time_series				
5	index_dim_count	1				
6						
7	node	Bastusel_upper	Bergnäs_upper	Båtfors_upper	Finnfors_upper	Gallejaur_upper
8	alternative	Base	Base	Base	Base	Base
9	index	fix_node_state	fix_node_state	fix_node_state	fix_node_state	fix_node_state
10						
11	2019-01-01T00:00:00	5581,44	114543,6	1117,2	234	1224
12	2019-01-01T01:00:00	nan	nan	nan	nan	nan
13	2019-01-07T23:00:00	5417,28	105898,8	891,1	234	2808

JSON format

The JSON export is complete since it contains all of the data from the database. The JSON format consists of a single JSON object with the following `OPTIONAL` keys:

- **entity_classes**: the value of this key `MUST` be a JSON array, representing a list of entity classes. Each element in this array `MUST` be itself a JSON array and `MUST` have three elements:
  - The first element `MUST` be a JSON string, indicating the entity class name.
  - The second element `MUST` be a JSON array, indicating the member entity classes. Each element in this array `MUST` be a JSON string, indicating the entity class name. In case of 0-D entity class, the array is empty.
  - The third element `MUST` be either a JSON string, indicating the entity class description, or null.

  - The fourth element MUST be either a JSON integer, indicating the entity class icon code, or null.
  - The fourth element MUST be a JSON boolean, indicating the state of active by default.
- **superclass_subclasses**: the value of this key MUST be a JSON array, representing a list of superclasses. Each element in this array MUST be itself a JSON array and MUST have two elements:
  - The first element MUST be a JSON string, indicating the superclass name.
  - The second element MUST be a JSON string, indicating the subclass name.
- **entities**: the value of this key MUST be a JSON array, representing a list of entities. Each element in this array MUST be itself a JSON array and MUST have three elements:
  - The first element MUST be a JSON string, indicating the entity class name.
  - The second element MUST be a JSON array, if the entity is N-dimensional. In this case each element in the array MUST be a JSON string itself, each being an element of the entity. If the entity class is 0-D, this element MUST be a JSON string, indicating the entity name.
  - The third element MUST be either a JSON string, indicating the entity description, or null.
- **Entity alternatives**: the value of this key MUST be a JSON array, representing a list of entity alternatives. Each element in this array MUST be itself a JSON array and MUST have four elements:
  - The first element MUST be a JSON string, indicating the entity class name.
  - The second element MUST be either a JSON array or a JSON string. In the case of a N-dimensional entity the array MUST itself contain JSON strings representing the element name list of the entity. If the entity is 0-D, a JSON string of the name of the entity is enough, but also a JSON array of one element is supported.
- **entity_groups**: the value of this key MUST be a JSON array, representing a list of entity groups. Each element in this array MUST be itself a JSON array and MUST have three elements:
  - The first element MUST be a JSON string, indicating the entity class.
  - The second element MUST be a JSON string, indicating the entity group name.
  - The third element MUST be a JSON string, indicating the member entity's name.
- **parameter_value_lists**: the value of this key MUST be a JSON array, representing a list of parameter value lists. Each element in this array MUST be itself a JSON array and MUST have two elements:
  - The first element MUST be a JSON string, indicating the parameter value list name.
  - The second element MUST be either a JSON object, string, number, or null, indicating the value.
- **parameter_definitions**: the value of this key MUST be a JSON array, representing a list of parameter definitions. Each element in this array MUST be itself a JSON array and MUST have five elements:
  - The first element MUST be a JSON string, indicating the entity class name.
  - The second element MUST be a JSON string, indicating the parameter name.
  - The third element MUST be either a JSON object, string, number, or null, indicating the parameter default value.
  - The fourth element MUST be a JSON string, indicating the associated parameter value list, or null.
  - The fifth element MUST be either a JSON string, indicating the parameter description, or null.
- **parameter_values**: the value of this key MUST be a JSON array, representing a list of entity parameter values. Each element in this array MUST be itself a JSON array and MUST have four elements:
  - The first element MUST be a JSON string, indicating the entity class name.

  - The second element MUST be a JSON array, if the entity is N-dimensional. In this case each element in the array MUST be a JSON string itself, each being an element of the entity. If the entity class is 0-D, this element MUST be a JSON string, indicating the entity name.
  - The third element MUST be a JSON string, indicating the parameter name.
  - The fourth element MUST be either a JSON object, string, number, or null, indicating the parameter value.

  There is one OPTIONAL element:

  - The fifth element MUST either be a JSON string indicating the alternative, or null. If this element is not present, an alternative named Base will be created if it doesn't exist and the values will be set in that alternative.

- **alternatives**: the value of this key MUST be a JSON array, representing a list of alternatives. Each element in this array MUST be itself a JSON array and MUST have two elements:
  - The first element MUST be a JSON string, indicating the alternative name
  - The second element MUST be either a JSON string, indicating the alternative description, or null.
- **scenarios**: the value of this key MUST be a JSON array, representing a list of alternatives. Each element in this array MUST be itself a JSON array and MUST have two elements:
  - The first element MUST be a JSON string, indicating the scenario name.
  - The second element MUST be a JSON boolean, indicating the scenario alternative active state.
  - The third element MUST be either a JSON string, indicating the scenario description, or null.
- **scenario alternatives**: the value of this key MUST be a JSON array, representing a list of alternatives. Each element in this array MUST be itself a JSON array and MUST have three elements:
  - The first element MUST be a JSON string, indicating the scenario name.
  - The second element MUST be a JSON string, indicating the alternative name aowfiuhwaofiajw.

Example:

```
{
  "entity_classes": [
    ["connection",[],"A transfer of commodities between nodes. E.g. electricity line,gas␣
↪pipeline...",280378317271233,true],
    ["node",[],"A universal aggregator of commodify flows over units and connections,
↪with storage capabilities.",280740554077951,true],
    ["unit",[],"A conversion of one/many comodities between nodes.",281470681805429,
↪true],
    ["unit__from_node",["unit","node"],"Defines the `nodes` the `unit` can take input␣
↪from,and holds most `unit_flow` variable specific parameters.",281470681805657,true],
    ["unit__to_node",["unit","node"],"Defines the `nodes` the `unit` can output to,and␣
↪holds most `unit_flow` variable specific parameters.",281470681805658,true],
    ["connection__node__node",["connection","node","node"],"Holds parameters spanning␣
↪multiple `connection_flow` variables to and from multiple `nodes`.",null,true]
  ],
  "entities": [
    ["connection","Bastusel_to_Grytfors_disch",null],
    ["node","Bastusel_lower",null],
    ["node","Bastusel_upper",null],
    ["node","Grytfors_upper",null],
    ["unit","Bastusel_pwr_plant",null],
    ["unit__from_node",["Bastusel_pwr_plant","Bastusel_upper"],null],
```

(continues on next page)

(continued from previous page)

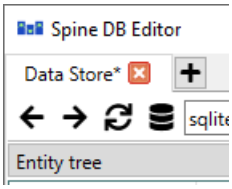

```
    ["unit__to_node",["Bastusel_pwr_plant","Bastusel_lower"],null],
    ["connection__node__node",["Bastusel_to_Grytfors_disch","Grytfors_upper","Bastusel_
↪lower"],null]
  ],
  "parameter_value_lists": [
    ["balance_type_list","balance_type_group"],
    ["balance_type_list","balance_type_node"],
    ["balance_type_list","balance_type_none"]
  ],
  "parameter_definitions": [
    ["connection","connection_availability_factor",1,null,"Availability of the␣
↪`connection`,acting as a multiplier on its `connection_capacity`. Typically between 0-
↪1."],
    ["connection__node__node","connection_flow_delay",{"type": "duration","data": "0h"},
↪null,"Delays the `connection_flows` associated with the latter `node` in respect to␣
↪the `connection_flows` associated with the first `node`."],
    ["node","balance_type","balance_type_node","balance_type_list","A selector for how␣
↪the `:nodal_balance` constraint should be handled."],
    ["node","demand",0,null,"Demand for the `commodity` of a `node`. Energy gains can be␣
↪represented using negative `demand`."],
    ["node","fix_node_state",null,null,"Fixes the corresponding `node_state` variable to␣
↪the provided value. Can be used for e.g. fixing boundary conditions."],
    ["node","has_state",null,null,"A boolean flag for whether a `node` has a `node_
↪state` variable."],
    ["unit__from_node","unit_capacity",null,null,"Maximum `unit_flow` capacity of a␣
↪single 'sub_unit' of the `unit`."],
    ["unit__to_node","unit_capacity",null,null,"Maximum `unit_flow` capacity of a single
↪'sub_unit' of the `unit`."]
  ],
  "parameter_values": [
    ["connection__node__node",["Bastusel_to_Grytfors_disch","Grytfors_upper","Bastusel_
↪lower"],"connection_flow_delay",{"type": "duration","data": "1h"},"Base"],
    ["node","Bastusel_upper","demand",-0.2579768519,"Base"],
    ["node","Bastusel_upper","fix_node_state",{"type": "time_series","data": {"2019-01-
↪01T00:00:00": 5581.44,"2019-01-01T01:00:00": -1,"2019-01-07T23:00:00": 5417.28}},"Base
↪"],
    ["node","Bastusel_upper","has_state",null,"Base"],
    ["unit__from_node",["Bastusel_pwr_plant","Bastusel_upper"],"unit_capacity",170,"Base
↪"]
  ],
  "alternatives": [
    ["Base","Base alternative"]
  ]
}
```

11.9 Committing and History

11.9.1 Committing

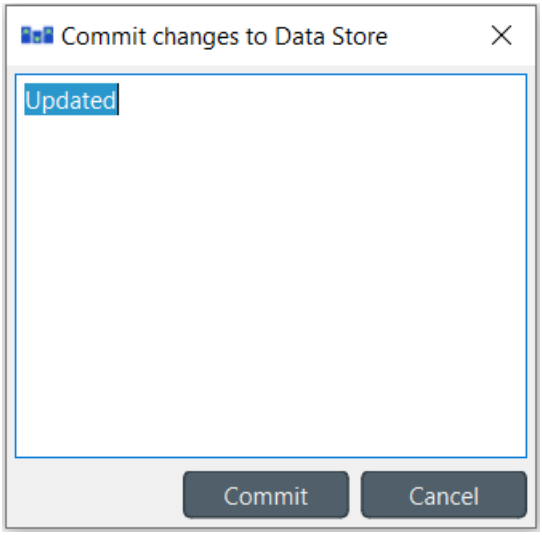
Note: Changes are not immediately saved to the database(s). They need to be committed separately.

An asterisk (*) in a tab of Spine Database Editor indicates that the session has uncommitted changes:



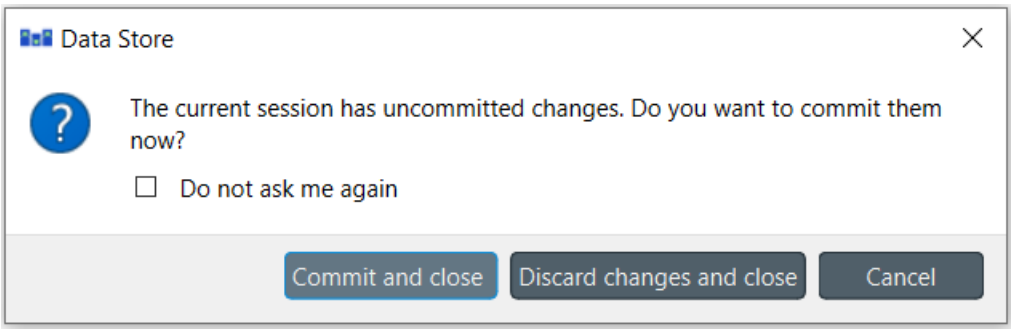
If the database is opened from a Data Store, the corresponding project item will also have a notification in its upper right corner indicating that the database has uncommitted changes.

To commit your changes, select **Session -> Commit** from the hamburger menu or press **Ctrl+Enter** while the Spine Database Editor -window is active to open the commit dialog:



There is a default text "Updated" readily filled in. It is however good practise to write a short message that clearly describes what changes have been made to the database. Once the commit message is to you liking, press **Commit**. Any changes made in the current session will be saved into the database.

If you try to close a tab with uncommitted changes the following dialog will open:



There are a few different options: **Commit and close** will do exactly what it says, **Discard changes and close** will automatically rollback the changes to the last commit and then close the editor tab. **Cancel** just closes the dialog and allows you to work on the database again. If you check the box **Do not ask me again** and select any of the other options beside **Cancel**, the selection you made will be automatically presumed whenever you close a tab with uncommitted changes. This means that you will not see this dialog again but the changes will be automatically committed or not depending on the selection you made previously.

11.9.2 Rollback

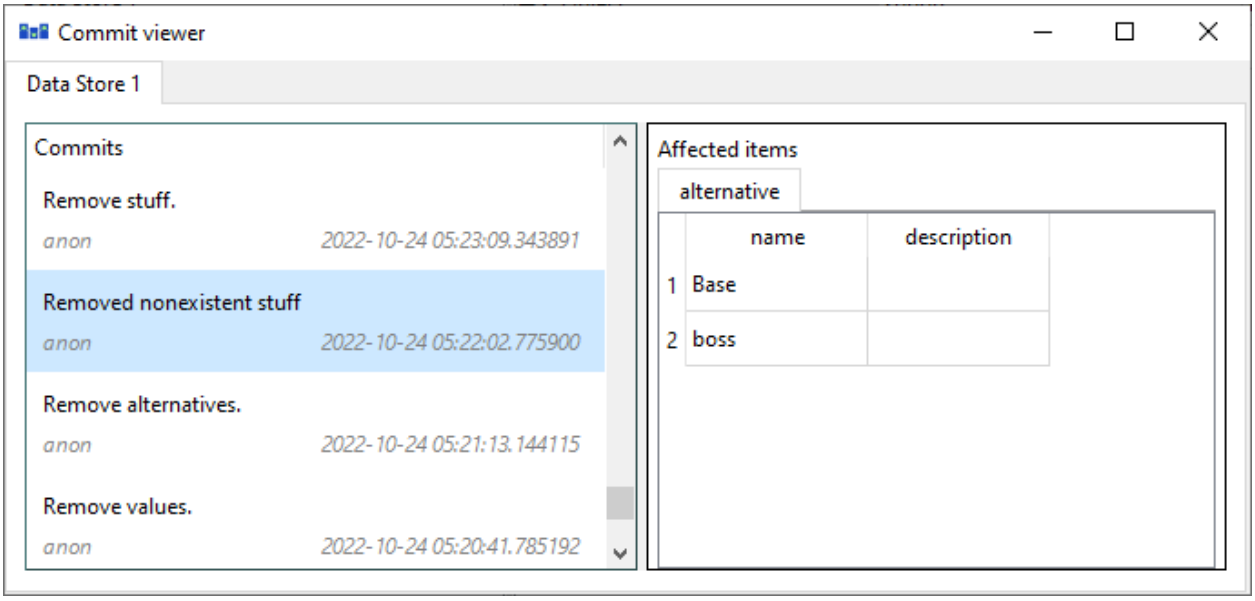
To undo *all* changes since the last commit, select **Session -> Rollback** from the hamburger menu or press **Ctrl+Backspace** while the Spine Database Editor -window is active. A dialog confirming the rollback action will open. From there, select **Rollback** to proceed.

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

Note: After rolling back to the last commit, the changes made in the session will no longer be available through undo/redo.

11.9.3 History

To examine the commit history of the database, select **Session -> History…** from the hamburger menu. The **commit viewer** will pop up:



All the commits made in the database will show here, each includes the timestamp. author and commit message. By selecting individual commits, the affected items can be inspected in the box on the right.

11.10 Vacuum

Vacuuming is available for Spine Databases in the SQLite format. Basically it tries to free up some unnecessary memory from the `.sqlite` -file. If you have very large databases, it might be beneficial to vacuum it once in a while. More detailed explanation on what vacuuming does to the SQLite database can be found here.

To vacuum a database, either press the Vacuum -button from the Data Store **Properties** -panel, or straight from the Spine Database Editors hamburger menu **Edit->Vacuum**.

After the vacuum is finished, a message informing the amount of bytes freed from the database is shown.

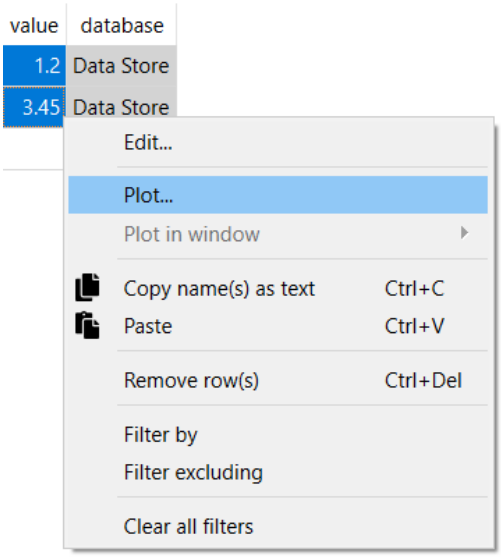
CHAPTER

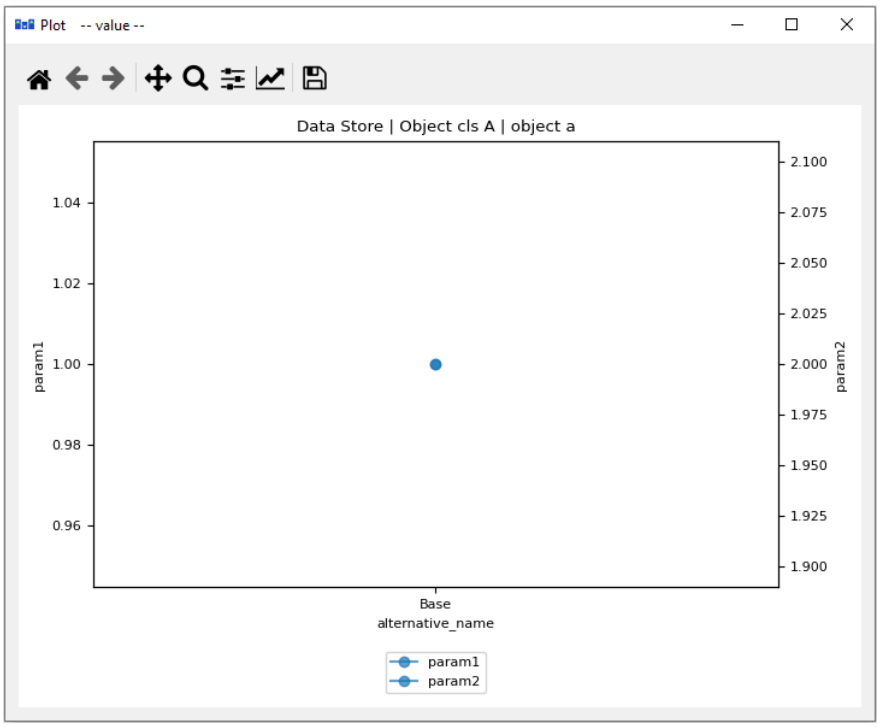
TWELVE

PLOTTING

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

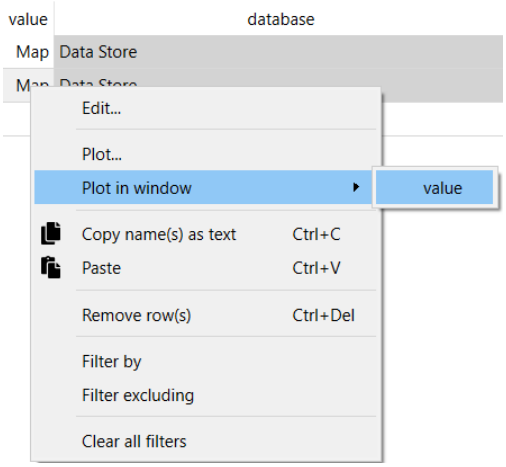
To plot a column, select the values from a table and then *Plot* from the **right click** popup menu.

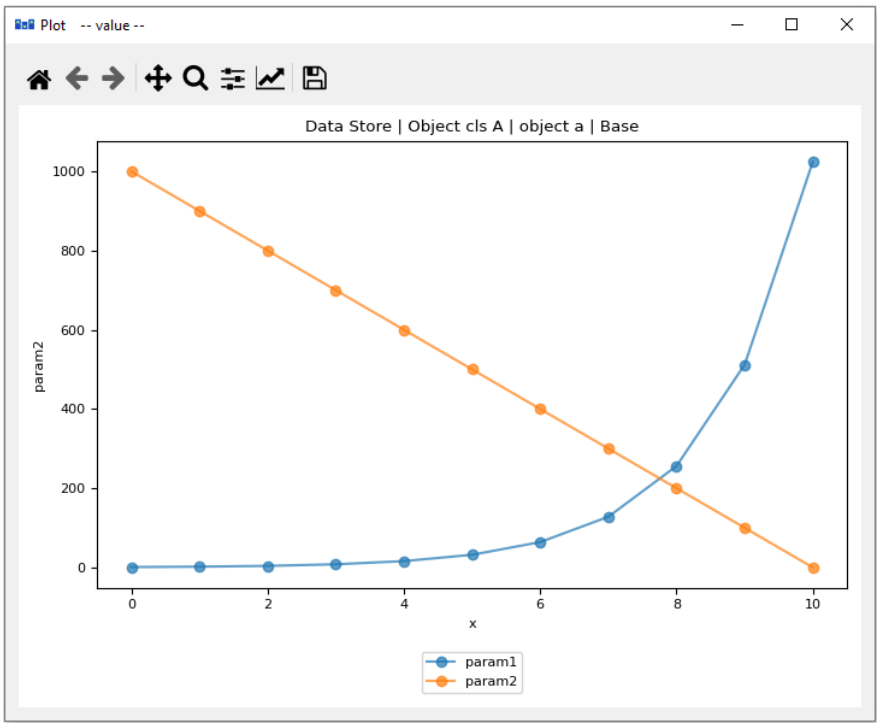




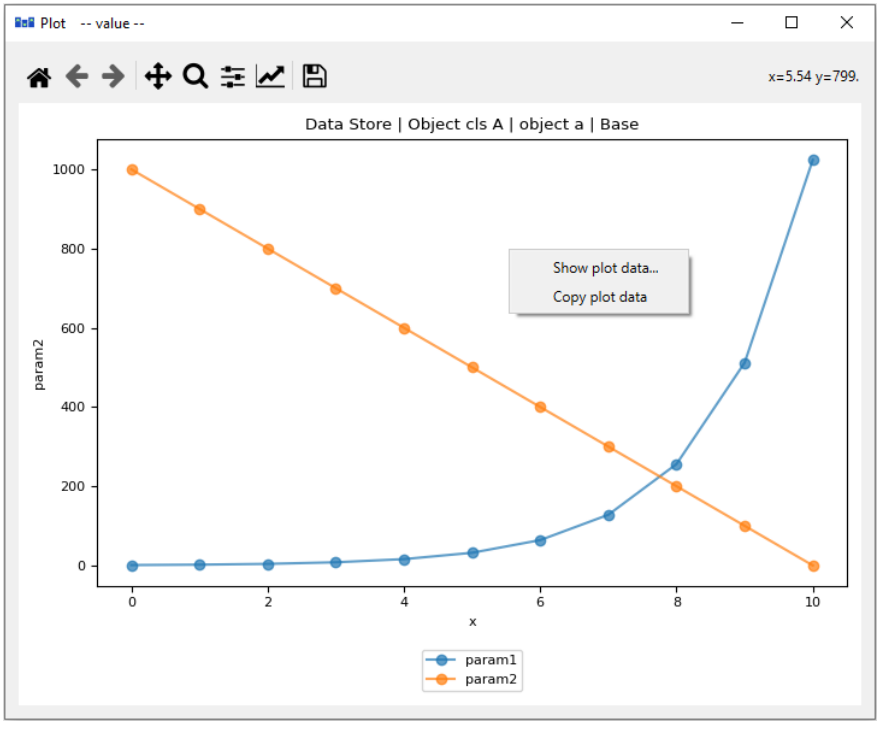
Selecting data in multiple columns plots the selection in a single window.

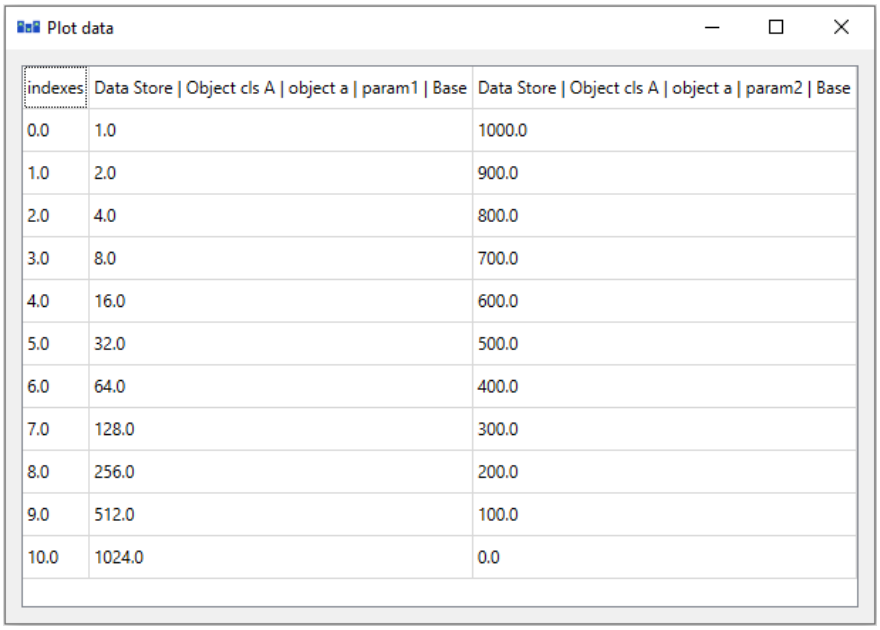
To add a plot to an existing window select the target plot window from the *Plot in window* submenu. There are some restrictions for what kinds of plots can be shown on the same window. In the example below two different maps have been plotted on the same graph.





If a plot is clicked with the right mouse button, options to copy or show the plot data are presented. When the data is copied it is saved to the clipboard in csv format with tab as the delimiter. If **Show plot data…** is clicked a new window opens that contains a table of the data used in the plot.



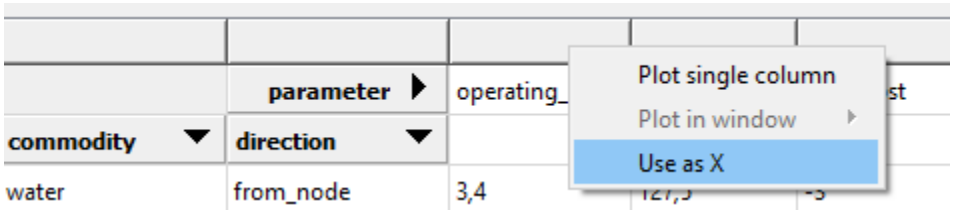


indexes	Data Store | Object cls A | object a | param1 | Base	Data Store | Object cls A | object a | param2 | Base
0.0	1.0	1000.0
1.0	2.0	900.0
2.0	4.0	800.0
3.0	8.0	700.0
4.0	16.0	600.0
5.0	32.0	500.0
6.0	64.0	400.0
7.0	128.0	300.0
8.0	256.0	200.0
9.0	512.0	100.0
10.0	1024.0	0.0

12.1 X column in pivot table

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

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



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

CHAPTER

THIRTEEN

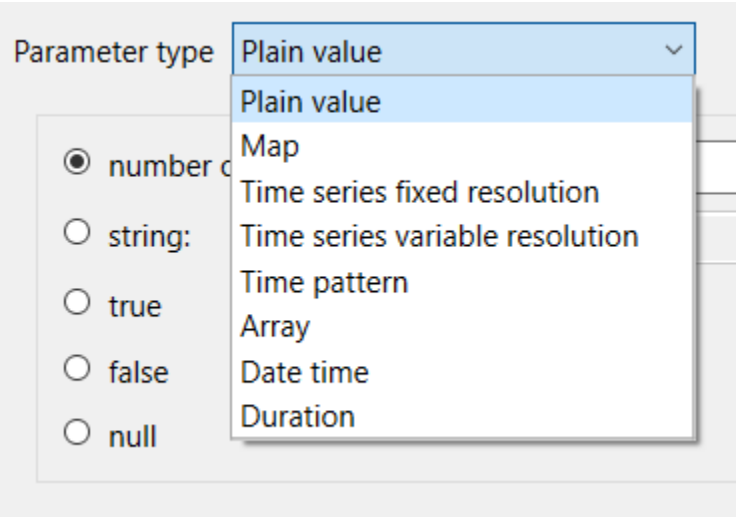
PARAMETER VALUE EDITOR

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

The editor can be opened by right clicking on the value field that you want to edit in a Spine database editor and selecting **Edit…** from the popup menu. In most cases the editor can be opened again by double clicking an existing value. With plain values like strings and floats this will not work since double clicking starts the text editor in that field.

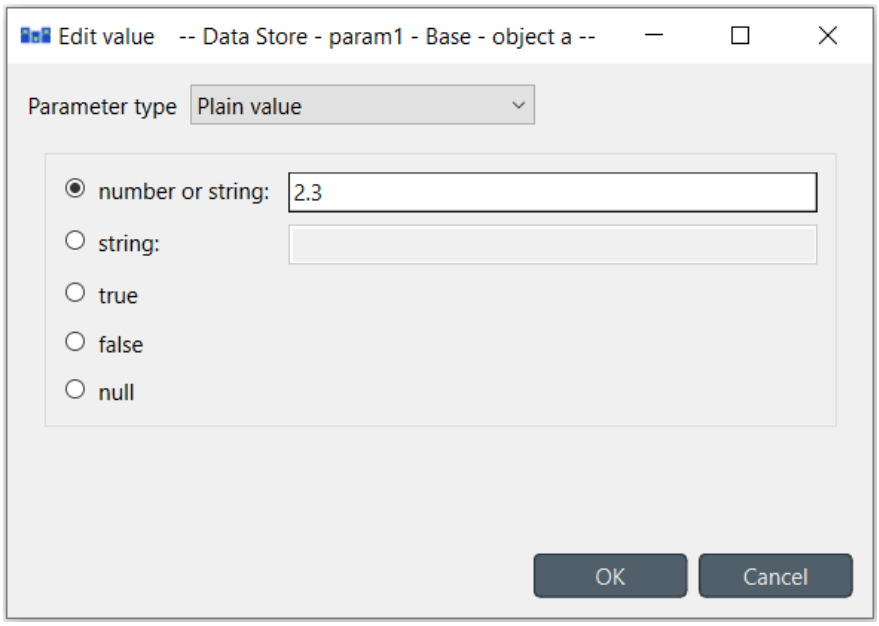
13.1 Choosing value type

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



13.2 Plain values

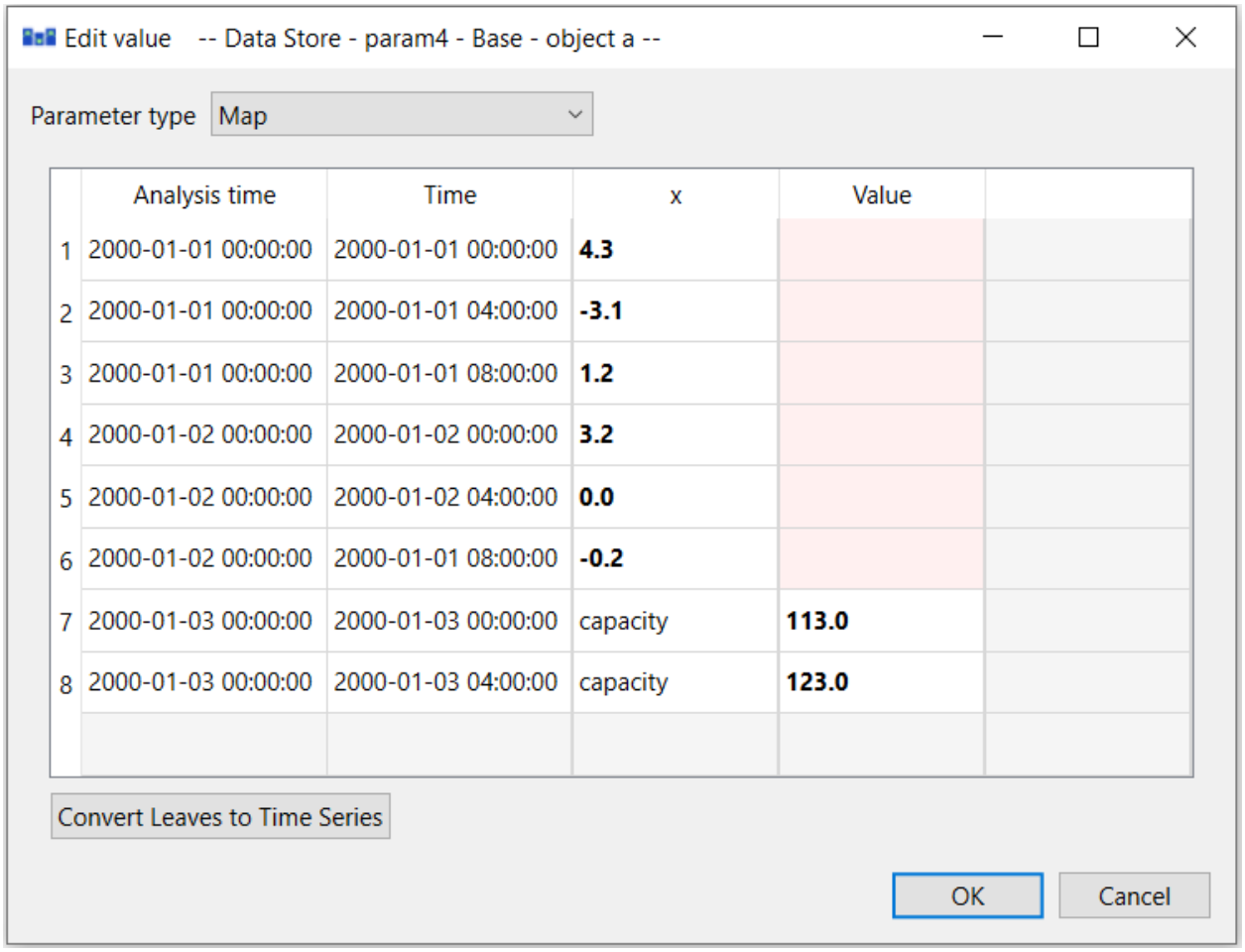
The simplest parameter values are of the *Plain value* type. The editor window lets you to write a number or string directly to the input field or set it to true, false or null as needed.



Numbers and strings can also be inserted without going through the value editor by double clicking on a value field. If the users input is a number like 3.14, the value type will be interpreted as *number or string*. If the input is a string like "ok", the value type will automatically be set to *string*.

13.3 Maps

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

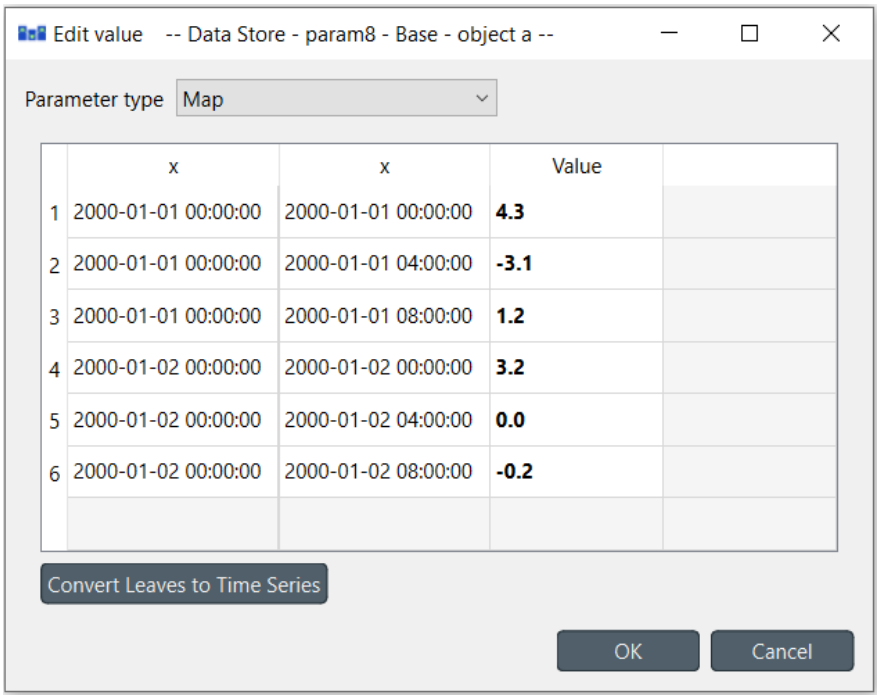


The extra gray column on the right allows expanding the map with a new dimension. You can append a value to the map by editing the bottom gray row. The reddish cells are merely a guide for the eye to indicate that the map has different nesting depths.

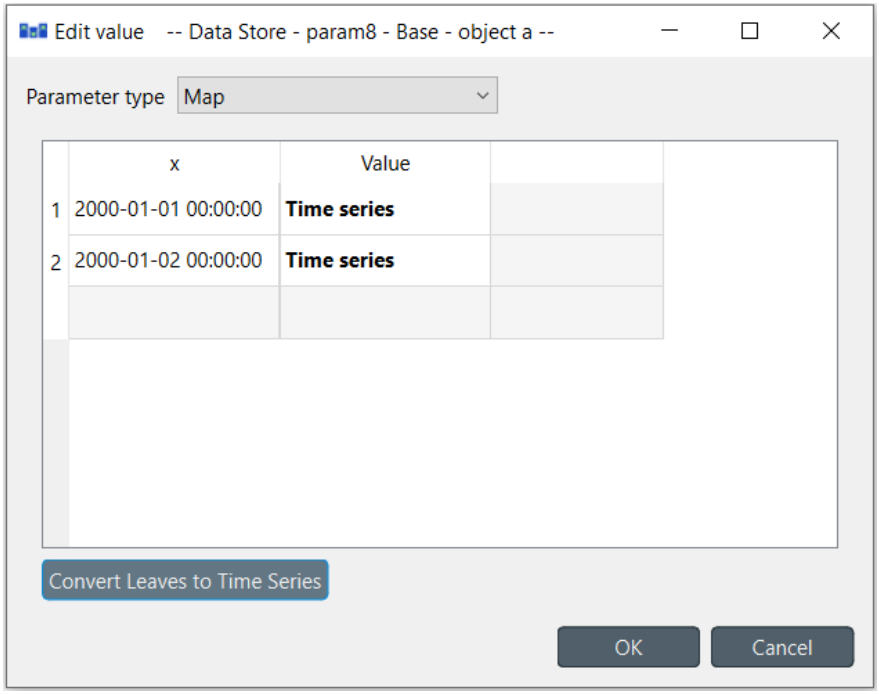
A **Right click** popup menu gives options to open a value editor for individual cells, plot individual and multiple cells, insert/remove rows or columns (effectively changing map's dimensions), trim empty columns from the right hand side and to copy and paste data. Copying and pasting data works between cells and external programs and can be also done using the usual **Ctrl+C** and **Ctrl+V** keyboard shortcuts.

The default name for new columns is *x*. Index names can however be modified. If a column holds both indices and data, the column header can also be modified. The last column of a map has to always contain values and therefore the header can't be modified from the default name *Value*.

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

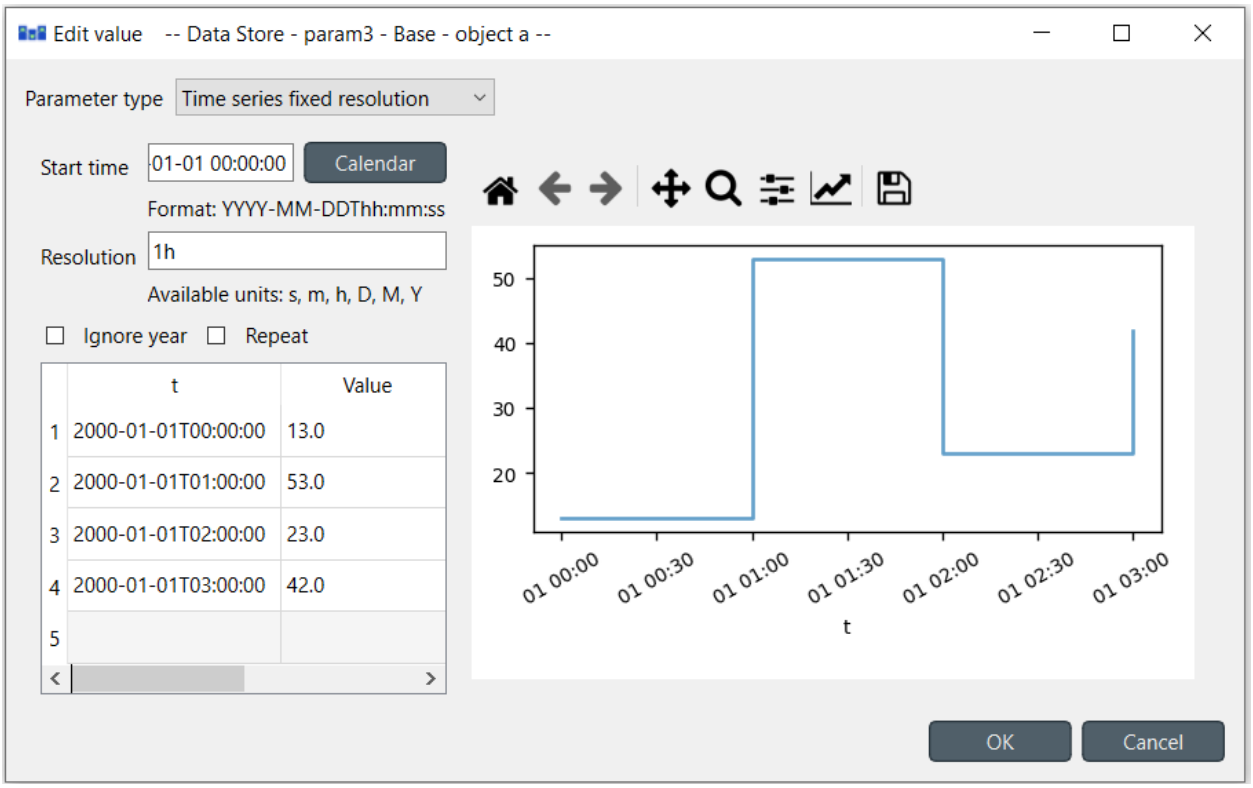


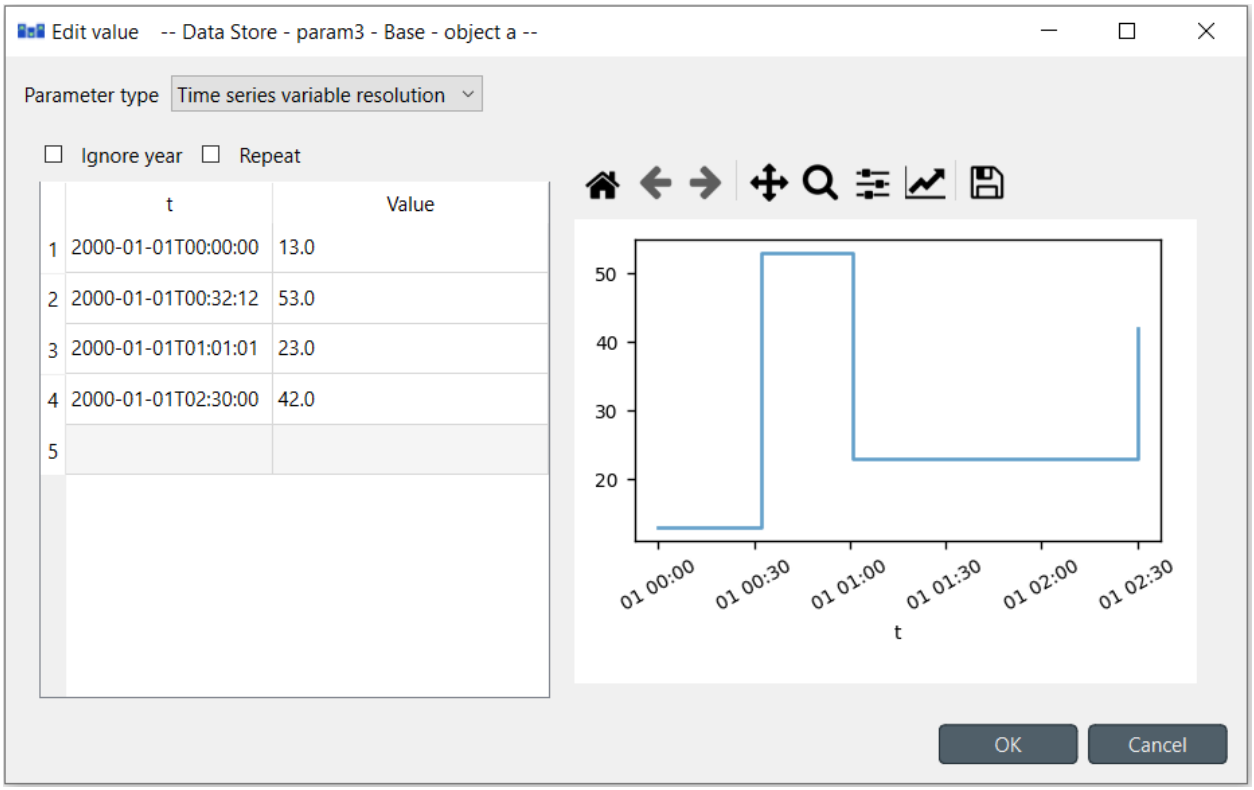
After clicking **Convert leaves to time series** the map looks like this:



13.4 Time series

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





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

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

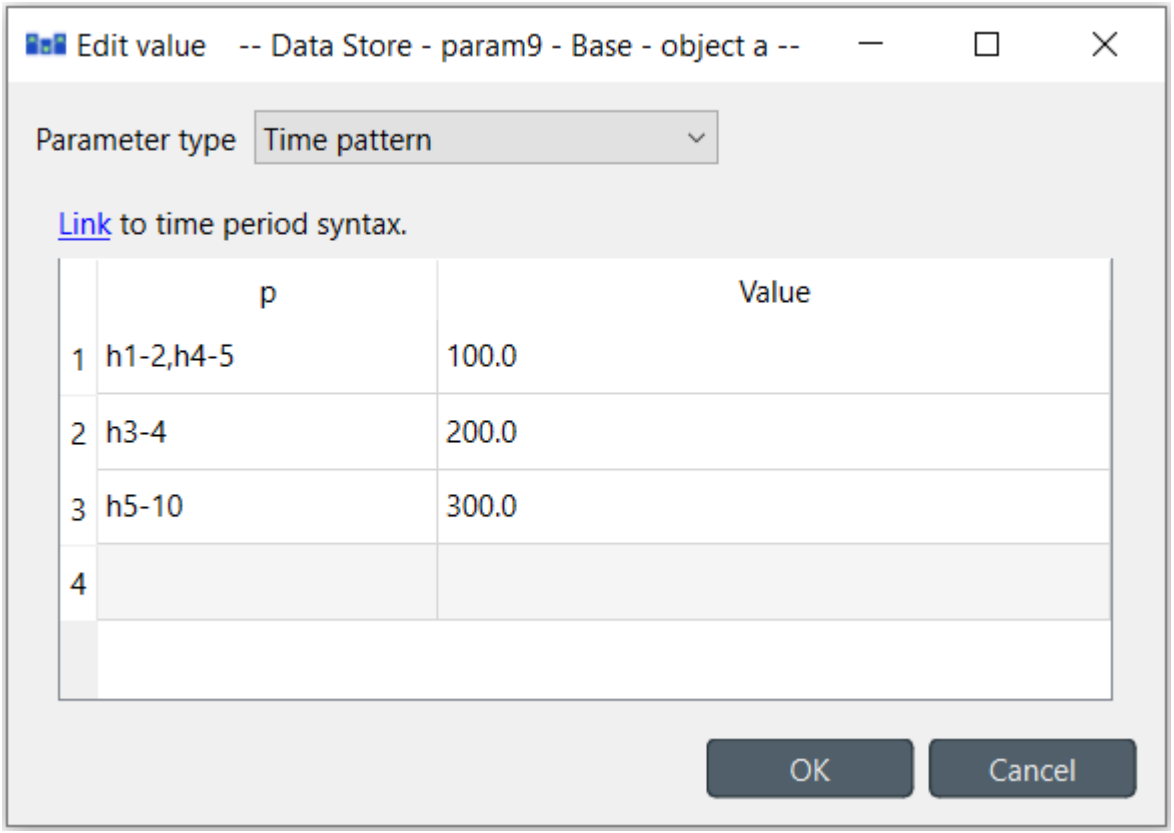
The time steps of a fixed resolution series are edited by the *Start time* and *Resolution* fields. The format for the start time is ISO8601. The *Resolution* field takes a single time step or a comma separated list of steps. If a list of resolution steps is provided then the steps are repeated so as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

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

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

13.5 Time patterns

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

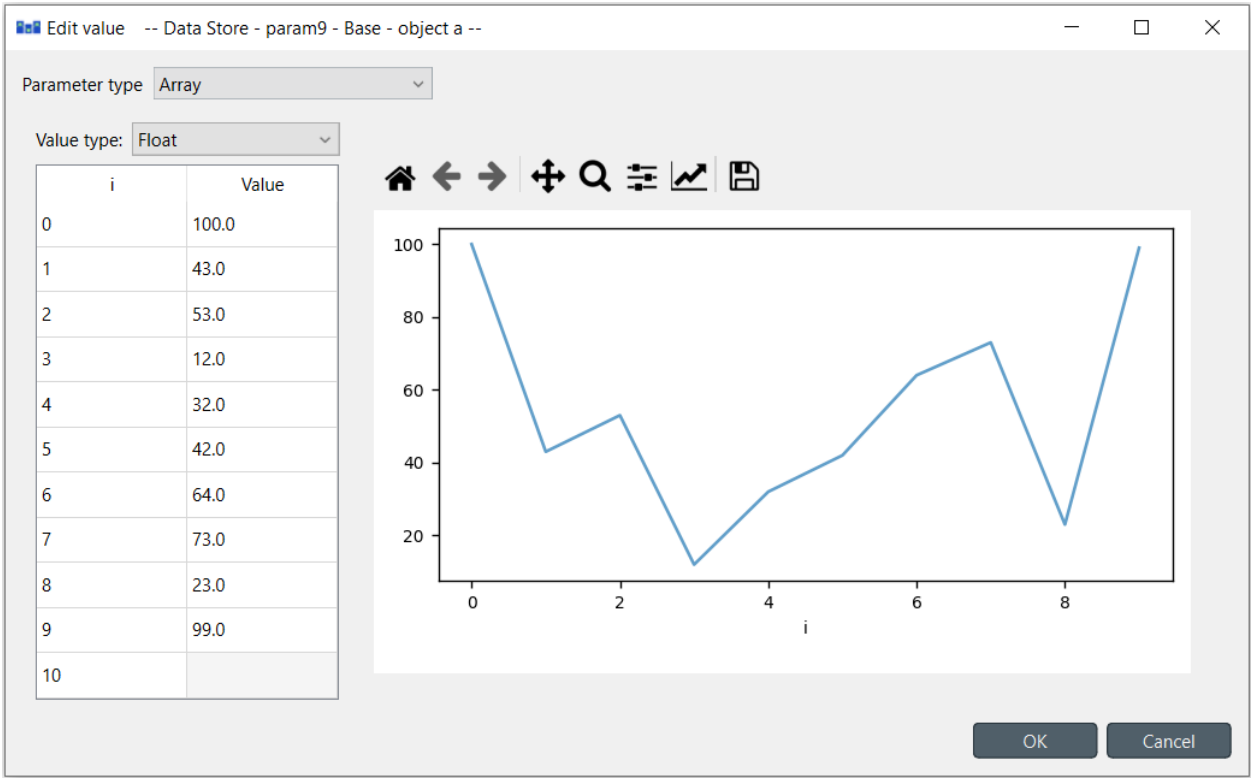


Time periods consist of the following elements:

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

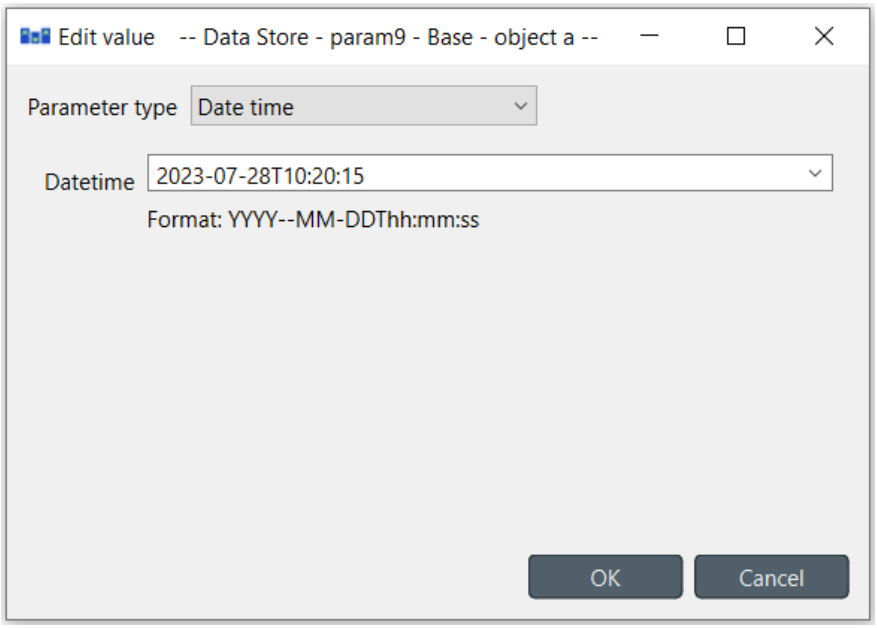
13.6 Arrays

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



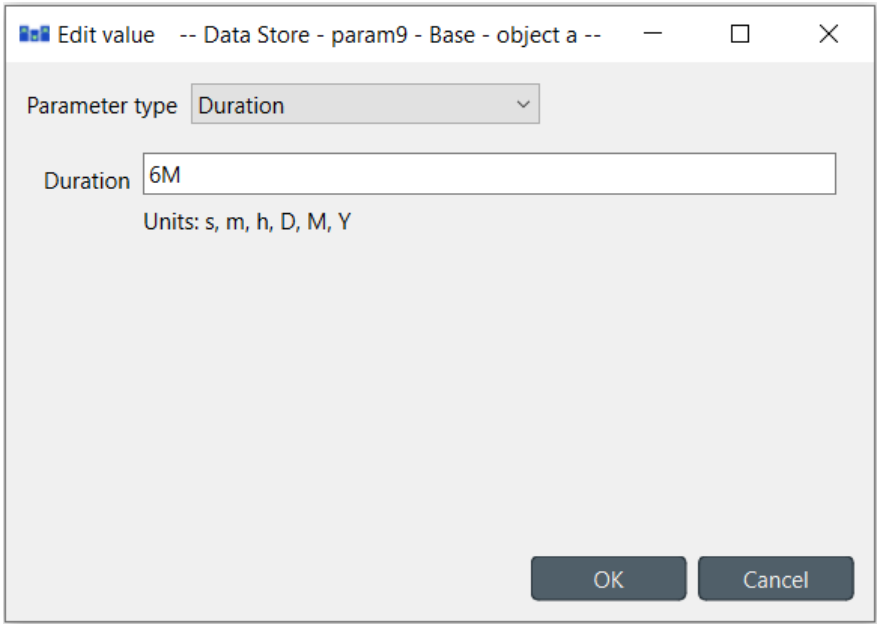
13.7 Datetimes

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



13.8 Durations

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



CHAPTER

FOURTEEN

SPINE METADATA DESCRIPTION

This is the metadata description for Spine, edited from https://frictionlessdata.io/specs/data-package/.

14.1 Required properties

`title`
One sentence description for the data sources.

`sources`
The raw sources of the data. Each source must have a `title` property and optionally a `path` property. Example:

```
"sources": [{
  "title": "World Bank and OECD",
  "path": "http://data.worldbank.org/indicator/NY.GDP.MKTP.CD"
}]
```

`contributors`
The people or organisations who contributed to the data. Must include `title` and may include `path`, `email`, `role` and `organization`. Example:

```
"contributors": [{
  "title": "Joe Bloggs",
  "email": "joe@bloggs.com",
  "path": "http://www.bloggs.com",
  "role": "author"
}]
```

Role is one of `author`, `publisher`, `maintainer`, `wrangler`, or `contributor`.

`created`
The date this data was created or put together, in ISO8601 format (YYYY-MM-DDTHH:MM)

14.2 Optional properties

description
A description of the data. Describe here how the data was collected, how it was processed etc.

The description MUST be markdown formatted – this also allows for simple plain text as plain text is itself valid markdown. The first paragraph (up to the first double line break) should be usable as summary information for the package.

spine_results_metadata
Key contains results metadata (described in a separate document).

keywords
An array of keywords

homepage
A URL for the home on the web that is related to this data package.

name
Name of the data package, url-usable, all-lowercase string.

id
Globally unique id, such as UUID or DOI

licenses
Licences that apply to the data. Each item must have a `name` property (Open Definition license ID) or a `path` property and may contain `title`. Example:

```
"licenses": [{
  "name": "ODC-PDDL-1.0",
  "path": "http://opendatacommons.org/licenses/pddl/",
  "title": "Open Data Commons Public Domain Dedication and License v1.0"
}]
```

temporal
Temporal properties of the data (if applicable). Example using DCMI Period Encoding Scheme:

```
"temporal": {
  "start": "2000-01-01",
  "end": "2000-12-31",
  "name": "The first year of the 21st century"
}
```

spatial
Spatial properties of the data (if applicable). Example using DCMI Point Encoding Scheme:

```
"spatial": {
  "east": 23.766667,
  "north": 61.5,
  "projection": "geographic coordinates (WGS 84)",
  "name": "Tampere, Finland"
}
```

unitOfMeasurement
Unit of measurement. Can also be embedded in `description`.

CHAPTER

FIFTEEN

IMPORTING AND EXPORTING DATA

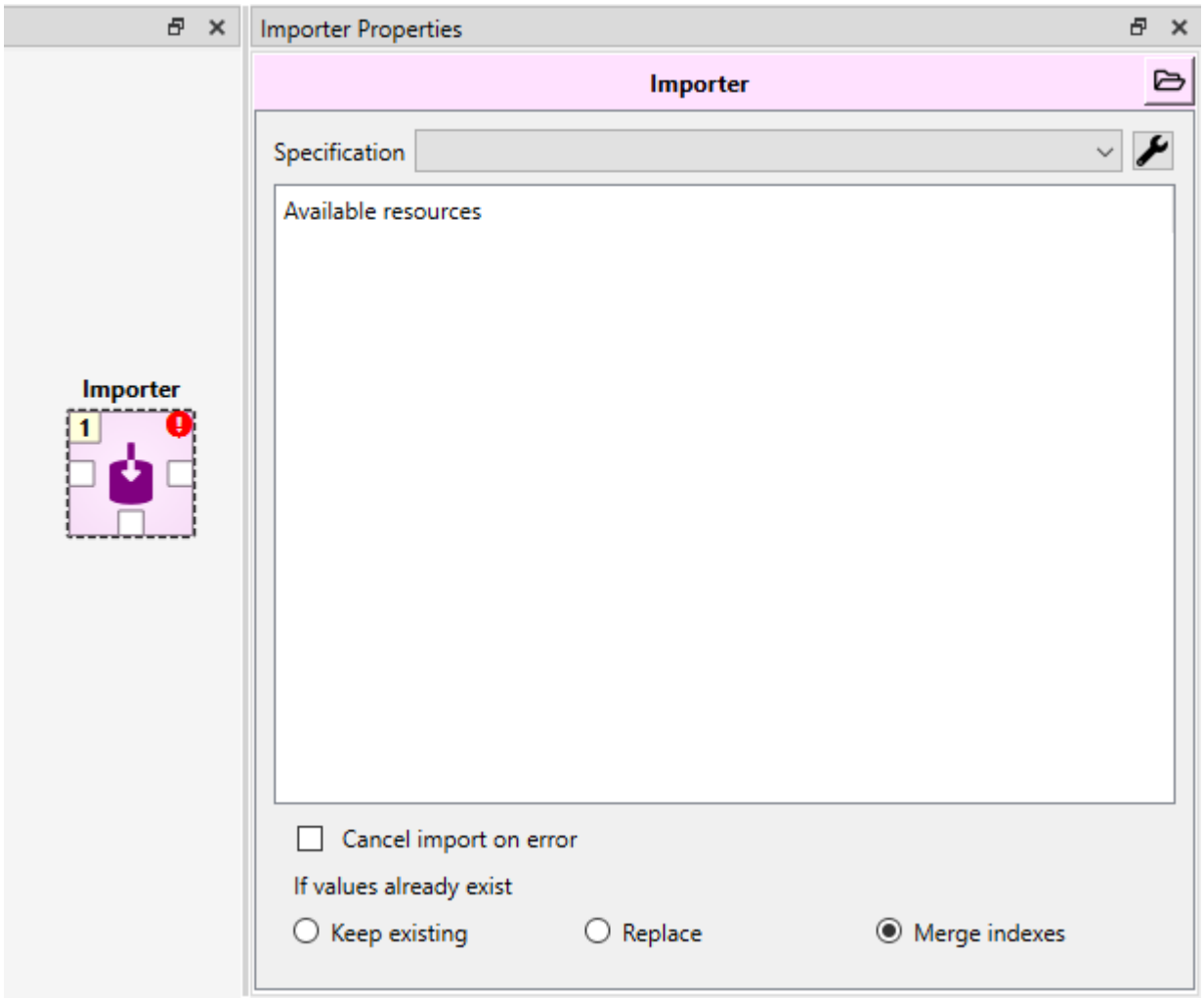
This section explains the different ways of importing and exporting data to and from a Spine database.

15.1 Importing data with Importer

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

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

The heart of Importer is the **Importer Specification Editor** window in which the mappings from source data to Spine database entities are set up. The editor window can be accessed by the button in Importer's Properties dock where existing specifications can also be selected or by double-clicking the item on the **Design View**.



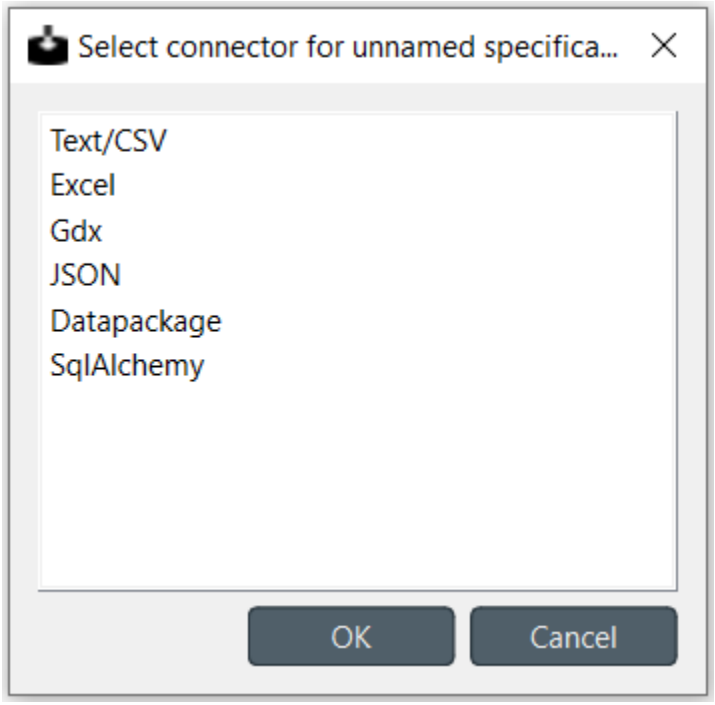
In the **Properties** tab there is also the option to cancel the import if an non-fatal error occurs during execution and also three options for how to handle if some of the values that you are trying to import already exist in the target Data Store. The three choices are:

Keep existing
: When this is selected, data that is already in the target Data Store will be kept and the new data the Importer tried to import is forgotten.

Replace
: This option means the imported data replaces old data in the target Data Store.

Merge indexes
: This option works mostly like the *Replace* option, but for indexed values like maps, time series and arrays the behavior is a little different. With these value types the old index-value pairs in the target Data Store will be kept and the new value pairs will be appended to it. If an index that is going to get imported already exist, the new imported value will overwrite the old value.

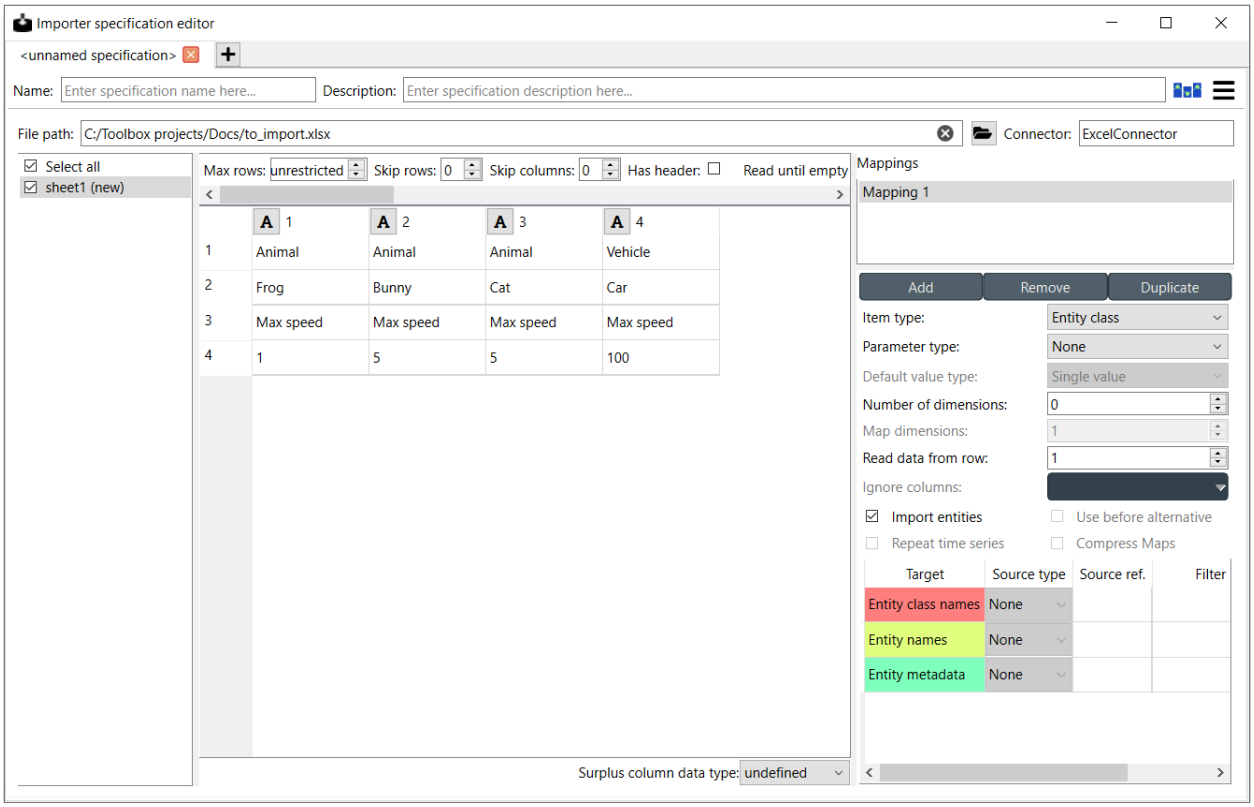
When the specification editor is opened for an Importer without a specification, a list of the supported connectors (file formats or other data sources) is presented. If the Importer is already connected to some data, it may have already selected the proper connector and is only waiting confirmation



When creating a Importer specification, it is usually helpful to have the data already connected to the Importer in question so it is easier to visualize how the importer is handling the data.

15.1.1 Importer specification editor

In the upper part of the specification editor, the name and description for the specification can be set. Underneath, the filepath is shown and it is allowed to br modified. Next to the filepath the used connector type is shown. On the left side, the different tables the file has or that user has set up are shown. Next to that there are the connector specific options and underneath that a preview of the selected table is shown. The items in the table are highlighted according to the selected mapping. The mappings are listed on the right side of the editor. underneath, the to be imported item types can be specified and other options set. Below that you can select the specific places in the source data where the entity names, values etc. will be taken from.



All tables available in the file selected in **File path** are listed in the leftmost dock widget. If the file does not have tables or the file type does not support them (e.g. CSV), all of the file's data will be in a single table called 'data'. The tables can be selected and deselected using the check boxes and only the selected ones will be imported. The option *Select All* is useful for selecting or deselecting all tables. If the Importer is opened in *anonymous mode*, there is also the option to add tables

---

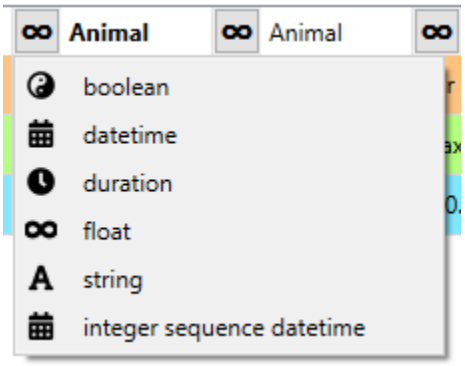
Tip: Multiple CSV files can be bundled into a *datapackage* which uses its own connector in Importer. Specifically, each CSV file in the datapackage shows up as a separate table in **Source tables**. See *Packing CSV files into datapackage* for more information on how to pack CSVs into a datapackage automatically within your workflow.

Next to the table dock widget, there is a small dock widget that allows to "format" the incoming data. The available options can differ depending on the selected connector. The above picture shows the available options for Excel files. **Max rows** specifies the amount of rows from the input data that are considered by the Importer. The option **Has header** converts the first row into headers. **Skip rows** and **Skip columns** skip the first *N* specified rows or columns from the table. If the table has empty rows or columns and some other data after that that you don't want to use, **Read until empty row/column on first column/row** options can be used to "crop" the imported data to the first relevant block of information. Other possible options for different connector types include **Encoding**, **Delimiter**, **Custom Delimiter**, **Quotechar** and **Maximum Depth**. **Load default mapping** sets all of the selections in the spec editor to their default values. Be careful not to press this button unless you want to wipe the whole specification clean.

Note: If you are working on a specification and accidentally press the *load default mapping* button you can undo previous changes for the specification from the hamburger menu or by pressing **Ctrl+Z**. To redo actions, or press **Crl+Y**.

When a table is selected, it's data and a preview of how the selected mapping will import the data will be presented under the options dock widget. An important aspect of data import is whether each item in the input data should be

read as a string, a number, a time stamp or something else. By default all input data is read as strings. However, more often than not things like parameter values are actually numbers. Though types are usually casted automatically, it is possible to manually control what type of data each column (and sometimes each row) contains from the preview table. Clicking the data type indicator button on column or row headers pops up a menu with a selection of available data types. Right clicking the column/row header also gives the opportunity to change the data type of all columns/rows at once.

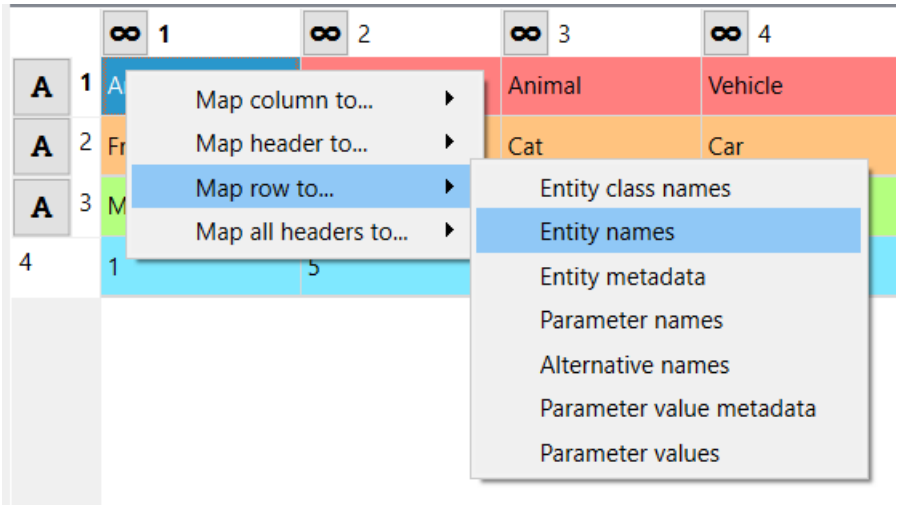


Under **Mappings** you can manage mappings by adding new ones and removing or duplicating existing ones. Each table has it's own mappings and every mapping has its own options. In **Mappings** you can select the mapping that you want to start modifying. Having multiple mappings for a single table allows to for example import multiple item types at the same time from a single table in a file.

Underneath **Mappings** there are options that help the importer get a feel for what kind of data it will be importing. The available *item type* options are *Entity class, Entity group, Alternative, Scenario, Scenario alternative* and *Parameter value list*. The other available options are dependent on the Item type. *Import entities* allows to import entities alongside or entity groups. *Parameter type* is used to specify what type of parameters, if any, the sheet contains. It has options *Value, Definition* and *None*. If *Value* or *Definition* is selected the value or respectively the default value type can be set from the drop-down list. *Use before alternative* is only available for *Scenario alternative* -item type. *Read data from row* lets you specify the row where the importer starts to read the data. *Ignore columns* allow you to select individual columns that you want to exclude from the whole importing process. *Number of dimensions* sets the amount of dimensions the entity to be imported has. *Repeat time series* sets the repeat flag to true when importing time series. *Map dimensions* sets the number of map indexes when importing map values. *Use before alternative* maps scenario before alternatives when importing scenario alternatives. *Compress Maps* can be used to compress value maps.

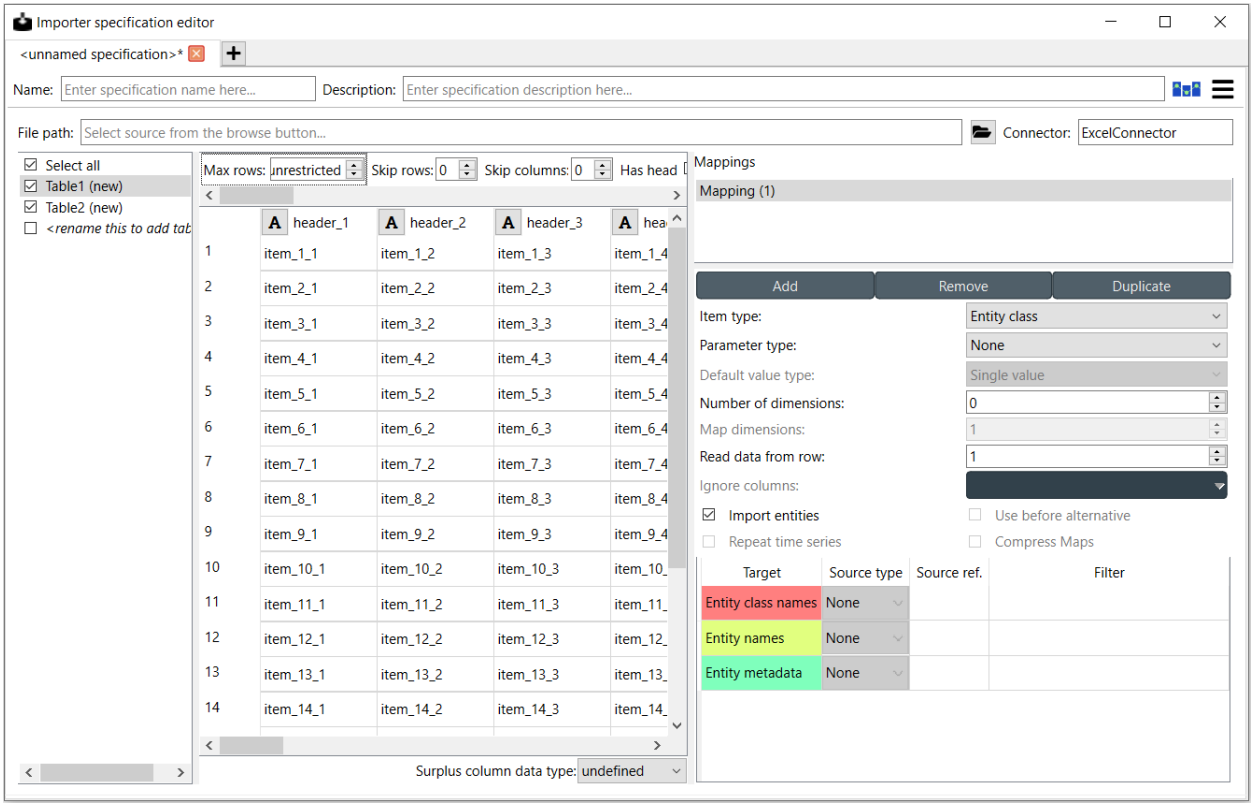
Once everything in the before mentioned options is in order, the next step is to set the mapping specification. Below the options there is the part where the decisions are made on how the input data is interpreted: which row or column contains the entity class names, parameter values, time stamps and so on. The dock widget contains all of the targets that the selected mapping options specify. Each target has a *Source reference* and a *Source type*. *Source type* specifies if the data for the target is coming in the form of a column, row, table name etc. In the *Source ref.* section you can pinpoint to the exact row, column etc. to use as the data. The *Filter* section can be used to further specify which values to include using regular expressions. More on regular expressions in section *Basic regular expressions for filtering*.

It might be helpful to fill in the *Source type* and *Source ref.* using the preview table in the *Sources data*. Right clicking on the table cells shows a popup menu that lets one to configure where the selected row/column/header is mapped to. It can also be used to simultaneously map all headers to one target.



15.1.2 Anonymous mode

The importer specification editor can be opened in a mode where there is no input data available. This might be useful when creating or modifying a generalized specifications. Anonymous mode entered when opening the specification of an Importer without incoming files or when opening the spec editor from Toolbox **Main Toolbar**.



In anonymous mode new tables can be created by double clicking *<rename this to add table>* and writing in a name for the new table. The preview will show an infinite grid of cells on which you can create different mappings.

Note: You can exit the Anonymous mode by browsing to and selecting an existing file using the controls in *File path*.

15.2 Exporting data with Exporter

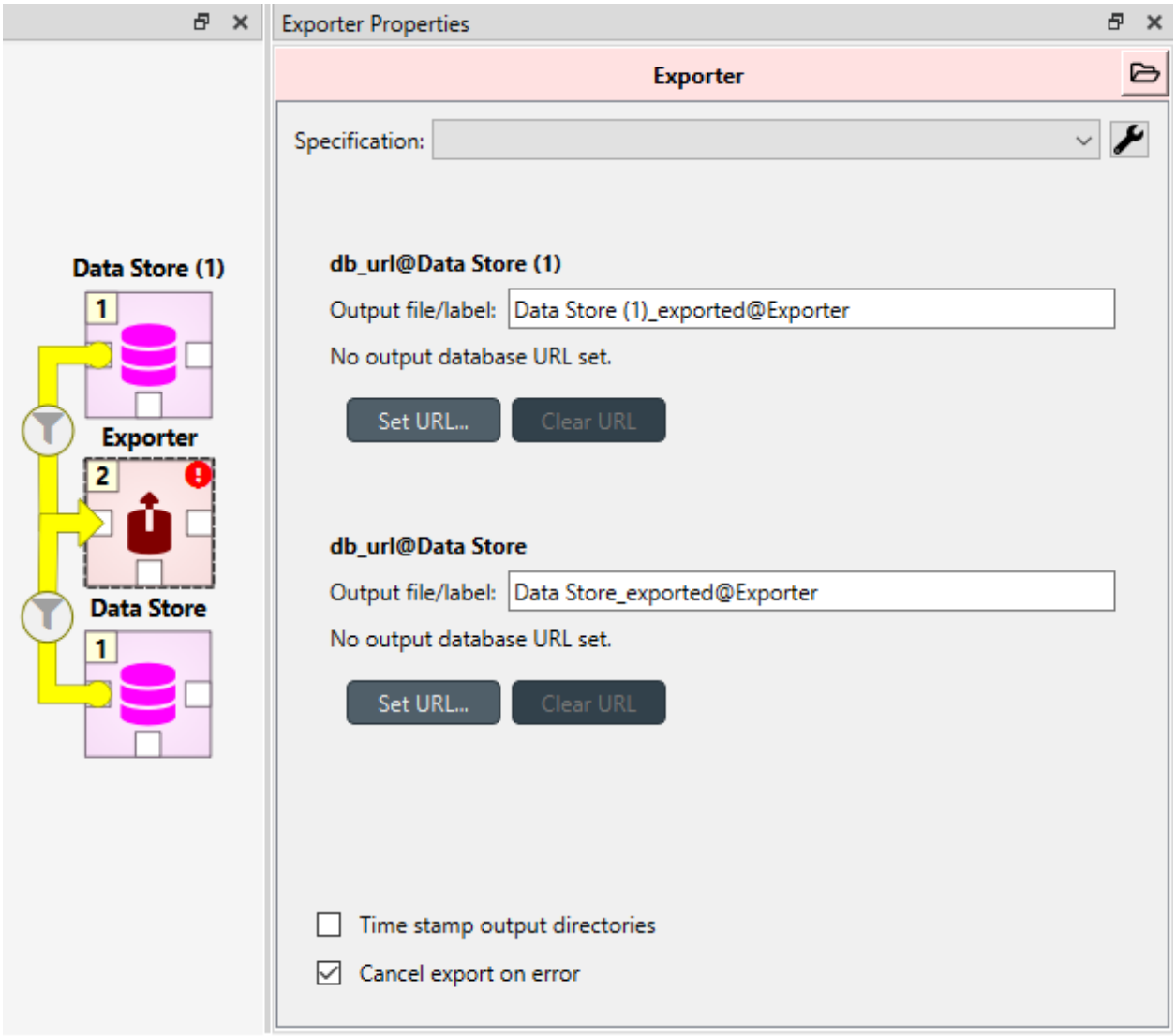
Exporter writes database data into regular files that can be used by Tools and external software that do not read the Spine database format. Various tabulated file formats are supported some of which require specific export settings; see below for more details.

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

Exporter also uses specifications so the same configurations can be reused by other exporters even in other projects. The specification can be edited in **Exporter specification editor** which is accessible by the button in the item's **Properties** dock or by double clicking Exporter's icon on the **Design View**. A specification that is not associated with any specific Exporter project item can be created and edited from the Main toolbar.

15.2.1 Properties dock

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



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

Data Stores that are connected to the exporter and are available for export are listed below the *Specification* combobox. An output label is required for each database and one Exporter can't have the same output label for two different Data Stores at the same time. Two different Exporters can have the same output label names since they are located in a different directories. The default label for the output files is of the format `<name of input Data Store>_exported@<name of Exporter>`.

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

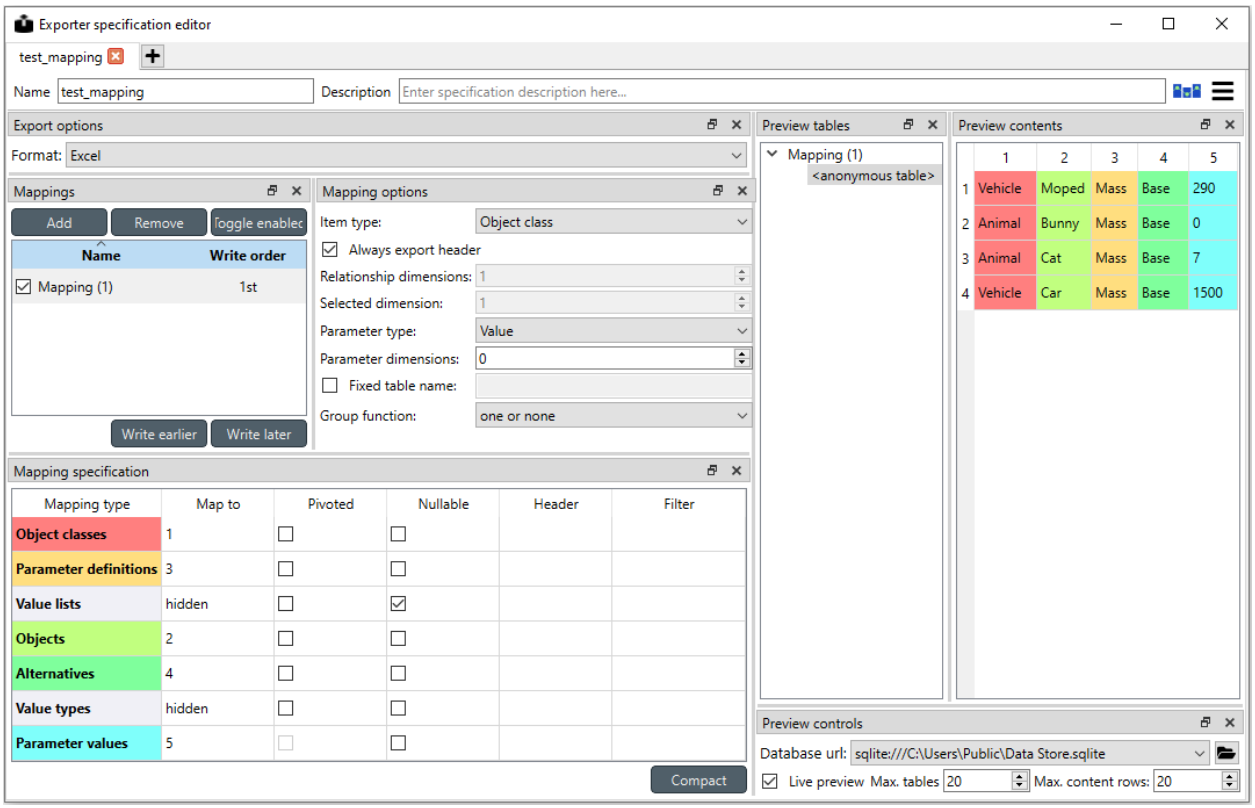
The *Cancel export on error* checkbox controls whether execution bails out on errors that may be otherwise non-fatal.

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

15.2.2 Exporter specification editor

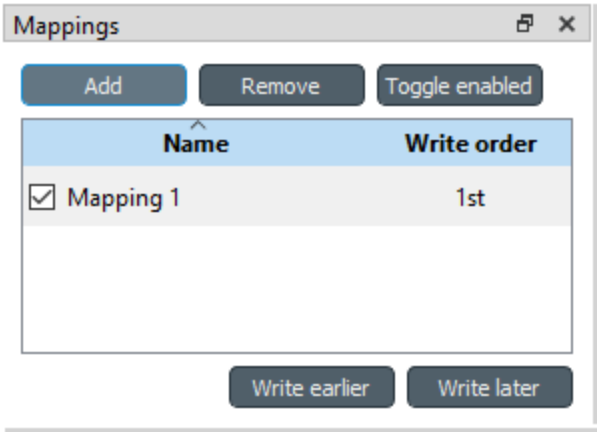
Specification editor is used to create **mappings** that define how data is exported to the output file. Mappings define one or more tables and their contents but are otherwise output format agnostic. Some output formats, e.g. SQL and gdx, interpret the tables in specific ways, however. Other formats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

When opened for the first time Specification editor looks like in the image below. The window is tabbed allowing multiple specifications to be edited at the same time. Each tab consists of dock widgets which can be reorganized to suit the user's needs. The 'hamburger' menu on the top right corner gives access to some important actions such as *Save* and *Close*. *Undo* and *redo* can be found from the menu as well. There is also a *Duplicate* option which creates a new tab in the spec editor that is otherwise the same but has no name and is missing the database url under *Preview controls*. This is handy if you want to create a new Exporter specification using an existing template instead of always starting form the beginning.

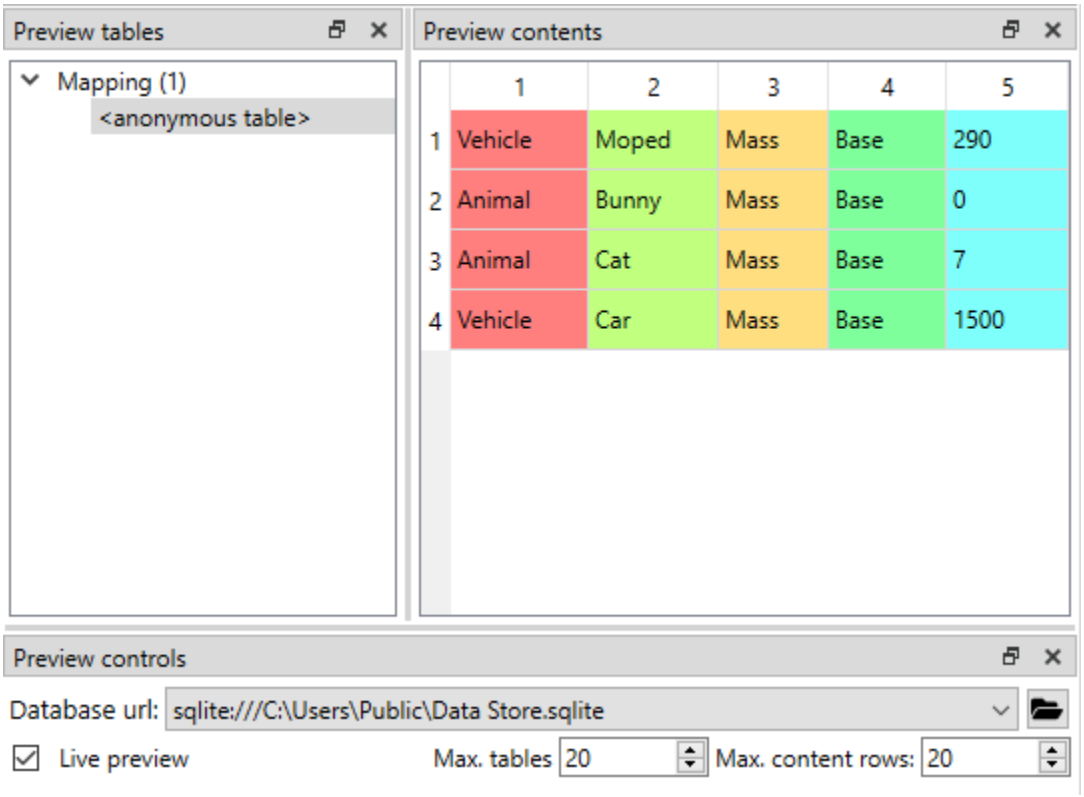


The only requirement for a specification is a name. This can be given on the *Name* field field on the top bar. The *Description* field allows for an additional explanatory text. The current output format can be changed by the *Format* combobox on *Export options* dock.

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



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

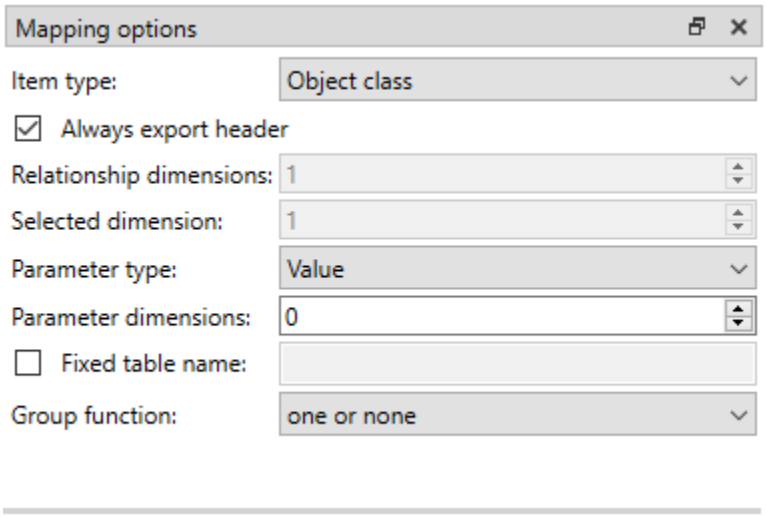


A preview of what will be written to the output is available in the preview dock widgets. To enable it, check the *Live preview* checkbox. A database connection is needed to generate the preview. The *Preview controls* dock provides widgets to choose an existing database or to load one from a file. Once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

Note: The preview is oblivious of any filters possibly set up in the workflow. Therefore, it may show entries, e.g. parameter values, that would be filtered out during execution.

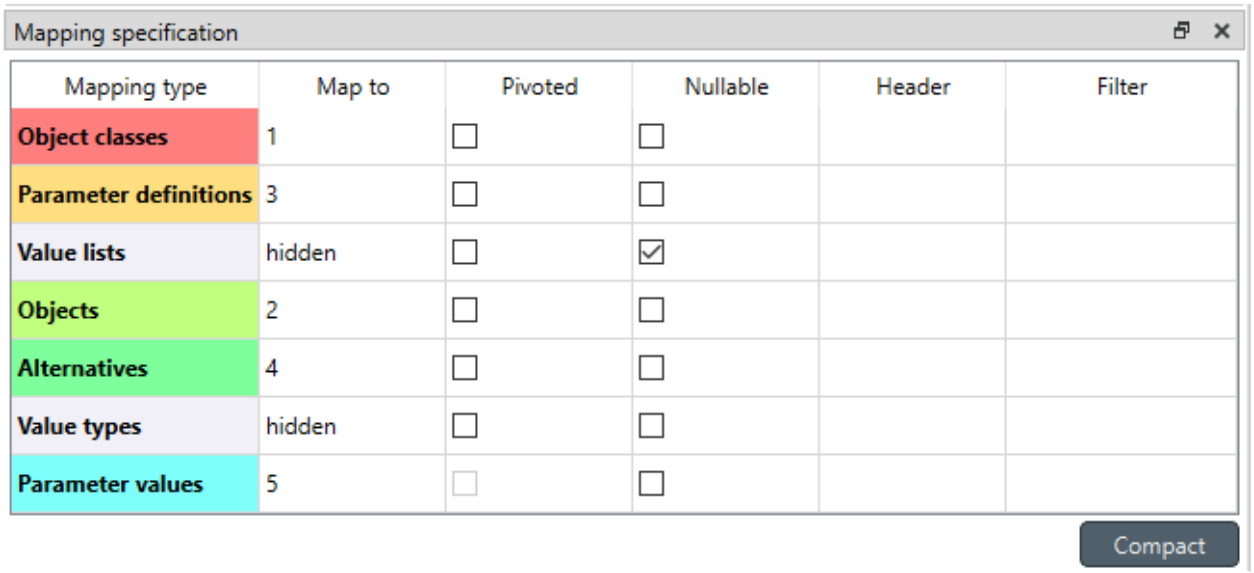
Mapping options

The currently selected mapping is edited using the controls in *Mapping options* and *Mapping specification* docks. The *Mapping options* dock contains controls that apply to the mapping as a whole, e.g. what data the output tables contain. It is important to choose *Item type* correctly since it determines what database items the mapping outputs and also dictates the mapping types that will be visible in the *Mapping specification* dock widget. It has options *Entity class, Entity class with dimension parameter, Entity group, Alternative, Scenario, Scenario alternative* and *Parameter value list*. The rest of the options besides *Group function* are item type specific and may not be available for all selections.



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

Mapping specification



Mapping specification contains a table which defines the structure of the mapping's output tables. Like mentioned before, the contents of the table depends on choices on *Mapping options*, e.g. the item type, parameter type or dimensions. Each row corresponds to an item in the database: entity class names, entity names, parameter values etc. The item's name is given in the *Mapping type* column. The colors help to identify the corresponding elements in the preview.

The *Map to* column defines the **position** of the item, that is, where the item is written or otherwise used when the output tables are generated. By default, a plain integral number in this column means that the item is written to that column in the output table. From the other choices, *hidden* means that the item will not show on the output. *Table name*, on the other hand, uses the item as output table names. For example, outputting entity classes as table names will generate one new table for every entity class in the database, each named after the class. Each table in turn will contain the parameters and entities of the table's entity class. If multiple mappings generate a table with a common name then each mapping appends to the same table in the order specified by the *Write order* column on *Mappings* dock.

The *column header* position makes the item a column header for a **buddy item**. Buddy items have some kind of logical relationship with their column header, for instance the buddy of an entity class is its entities; setting the entity class to *column header* will write the name of the class as the entity's column header.

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

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

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

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

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

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

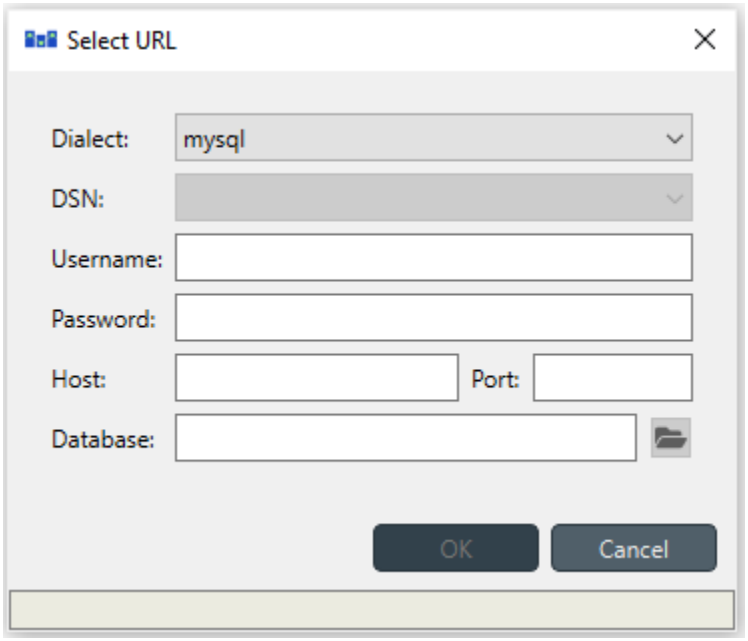
15.2.3 CSV and multiple tables

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

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

15.2.4 SQL export

To set up export to a remote database, first an Exporter specification with SQL selected as the format needs to be saved. The Exporter needs to also be connected to some input Data Store. From the Exporters **Properties** dock widget an output database can be specified for each input Data Store respectively by clicking the **Set URL...** button. A small new window opens with a few settings to set up the output database. Currently only mysql and sqlite are supported, even though mssql, postgresql and oracle are also listed as options for the dialect. Once a URL is set it can be removed by pressing the **Clear URL** button on the **Properties** tab.



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

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

- Pivot tables do not generally make sense with the SQL backend unless the resulting table somehow follows the above rules.

15.2.5 GAMS gdx export

Note: You need to have GAMS installed to use this functionality. However, you do not need to own a GAMS license as the demo version works just as well. See *Setting up Consoles and External Tools* for more information.

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

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

Sets:

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

Parameters:

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

Scalars:

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

15.3 Basic regular expressions for filtering

See regular expressions on wikipedia and on Python's documentation. Both the Exporter and Importer have applications for regular expressions in their respective *Mapping specifications* dock widgets. Below are examples on how to create some basic filters for these applications.

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

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

Excluding an item

While perhaps not the most suitable task for regular expressions it is still possible to 'negate' a filter. As an example, `^(?!node)` excludes all item names that start with 'node'.

CHAPTER

SIXTEEN

SPINE ENGINE SERVER

16.1 Notes

Here is a list of items that you should be aware of when running projects on Spine Engine Server.

- **Projects must be self-contained**. The project directory must contain all input and output files, file/db references, Specification files and scripts.
- **Work or Source directory execution mode** setting is ignored. Tools are always executed in 'source' directory, i.e. in the directory where the Tool Spec main script resides.
- **Python Basic Console**. Interpreter setting in Tool Specification Editor is ignored. Basic Console runs the same Python that was used in starting the Server.
- **Python Jupyter Console**. Kernel spec setting in Tool Specification Editor is ignored. Jupyter Console is launched using the **python3** kernel spec. This must be installed before the server is started. See instructions below.
- **Julia Basic Console**. Julia executable setting in Tool Specification Editor is ignored. Basic Console runs the Julia that is found in the server machine's PATH. See installation instructions below.
- **Julia Jupyter Console**. Kernel spec setting in Tool Specification Editor is ignored. Jupyter Console is launched using the **julia-1.8** kernel spec. This must be installed before the server is started. See instructions below.

16.2 Setting up Spine Engine Server

You can either install the entire Spine Toolbox or just the required parts to run the Spine Engine Server.

16.2.1 Minimal Installation

Spine Engine server does not need the entire Spine Toolbox installation. Only *spine-engine*, *spinedb-api* and *spine-items*. Note that the dependencies of *spine-items* are not needed. Here are the step-by-step instructions for a minimal installation:

1.1 Make a miniconda environment & activate it

1.2. Clone spine-engine

1.3. cd to *spine-engine* repo root, run:

```
pip install -e .
```

1.4. Clone spine-items

1.5. cd to *spine-items* repo root

1.6. Install *spine-items* **without dependencies** by running:

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

16.2.2 Full Installation

Install Spine Toolbox regularly

1.1. Make a miniconda environment & activate

1.2. Clone Spine Toolbox

1.3. Follow the installation instructions in README.md

16.2.3 Finalize Setting Up and Start Server

2. Create security credentials (optional)
   - cd to *<spine_engine_repo_root>/spine_engine/server/*
   - Create security certificates by running:

     ```
     python certificate_creator.py
     ```

   - The certificates are created into *<spine_engine_repo_root>/spine_engine/server/certs/* directory.
   - Configure allowed endpoints by creating file *<spine_engine_repo_root>/spine_engine/server/connectivity/certs/allowEndpoi*
   - Add IP addresses of the remote end points to the file
3. Install IPython kernel spec (*python3*) to enable Jupyter Console execution of Python Tools
   - Run:

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

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

     ```
     add IJulia
     ```

   - This installs *julia-1.8* kernel spec to *~/.local/share/jupyter/kernels* on Ubuntu or to *%APPDATA%jupyterkernels* on Windows
6. Start Spine Engine Server
   - cd to *<spine_engine_repo_root>/spine_engine/server/*
   - Without security, run:

     ```
     python start_server.py 50001
     ```

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

```
python start_server.py 50001 StoneHouse ./certs
```

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

Note: Valid port range is 49152-65535.

16.3 Setting up Spine Toolbox (client)

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

CHAPTER

SEVENTEEN

TERMINOLOGY

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

- **Arc** Graph theory term. See *Connection*.
- **Case study** Spine project has 13 case studies that help to improve, validate and deploy different aspects of the SpineOpt.jl and Spine Toolbox.
- **Connection** (aka **Arrow**) an arrow on Spine Toolbox Design View that is used to connect project items to each other to form a DAG.
- **Data Connection** is a project item used to store a collection of data files that may or may not be in Spine data format. It facilitates data transfer from original data sources e.g. spreadsheet files to Spine Toolbox. The original data source file does not need to conform to the format that Spine Toolbox is capable of reading, since there we can use an interpreting layer (Importer) between the raw data and the Spine format database (Data Store).
- **Data Package** is a data container format consisting of a metadata descriptor file (`datapackage.json`) and resources such as data files.
- **Data sources** are all the original, unaltered, sources of data that are used to generate necessary input data for Spine Toolbox tools.
- **Data Store** is a project item. It's a Spine Toolbox internal data container which follows the Spine data model. A data store is implemented using a database, it may be, for example, an SQL database.
- **Design View** A *sub-window* on Spine Toolbox main window, where project items and connections are visualized.
- **Direct predecessor** Immediate predecessor. E.g. in DAG *x*->*y*->*z*, direct predecessor of node *z* is node *y*. See also predecessor.
- **Direct successor** Immediate successor. E.g. in DAG *x*->*y*->*z*, direct successor of node *x* is node *y*. See also successor.
- **Directed Acyclic Graph (DAG)** Finite directed graph with no directed cycles. It consists of vertices and edges. In Spine Toolbox, we use project items as vertices and connections as edges to build a DAG that represents a data processing chain (workflow).
- **Edge** Graph theory term. See *Connection*
- **Element** is what the entities making up a multi dimensional entity are called. See also multidimensional entity.
- **Importer** is a project item that can be used to import data from e.g. an Excel file, transform it to Spine data structure, and into a Data Store.
- **Loop** (aka **jump**) is a special sort of connection which only connects the two attached project items if the user defined loop condition is met.
- **Multidimensional entity/entity class** (aka N-D entity/class) An entity/entity class that consists of multiple other entities that are as it's members. Acts just like any other entity/entity class.

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

CHAPTER

EIGHTEEN

CONTRIBUTION GUIDE

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

18.1 Reporting Bugs

18.1.1 Due Diligence

Before submitting a bug report, please do the following:

Perform basic troubleshooting steps.

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

18.1.2 What to Put in Your Bug Report

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

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

18.2 Feature Requests

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

18.3 Submitting features/bugfixes

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

18.3.1 Coding Style

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

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

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

18.3.2 Commit messages

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

A good example (inspired by [CB14])

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

18.3.3 Contributing to the User Guide

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

18.3.4 Contributing to the Spine Toolbox Graphical User Interface

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

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

18.3.5 Version Control Branching

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

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

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

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

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

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

18.3.6 Test-driven development is your friend

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

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

See *Unit Testing Guidelines* for more information.

18.3.7 Full example

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

Preparing your Fork

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

Making your Changes

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

Creating Pull Requests

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

18.4 References

CHAPTER

NINETEEN

DEVELOPER DOCUMENTATION

Here you can find developer specific documentation on Spine Toolbox.

19.1 UI Guidelines

19.1.1 Keyboard shortcuts

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

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

19.1.2 Action names

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

19.2 Unit Testing Guidelines

19.2.1 Test modules, directories

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

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

19.2.2 Running the tests

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

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

19.2.3 Helpers

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

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

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

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

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

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

19.3 Execution Tests

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

19.3.1 Executing the tests

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

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

19.4 Project Item Development

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

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

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

19.4.1 Getting started

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

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

19.4.2 Item info

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

19.4.3 Item Factory

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

19.4.4 Executable item

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

19.4.5 Toolbox side project item

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

19.4.6 Specifications

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

19.4.7 Toolbox GUI integration

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

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

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

19.4.8 Saving and restoring project items

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

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

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

19.4.9 Passing data between items: resources

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

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

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

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

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

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

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

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

[1] Data connection provides paths to local files.

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

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

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

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

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

19.4.10 Execution

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

19.5 Publishing to PyPI

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

19.5.1 Versioning of Spine Toolbox packages

Spine Toolbox packages use the latest Git tag to dynamically generate the version number. During the build process Git tags of the form `X.Y.Z` are sorted and the latest is used to generate the package version. If the tip of the current branch (`HEAD`) is at a tag, the version number is the tag. However, if there have been commits since the latest tag, the next version is guessed and a `dev??-*` component is included (e.g. `'0.7.0.dev77+gf9538fee.d20230816'`). Note that the `dev*` component also includes an indication of the number of commits since the last tag.

Under this scheme, the release process is simply to create a new Git tag, and publish it. However since the different Spine packages depend on each other, you need to update the different version number requirements in their respective `pyproject.toml` files. This can be done conveniently by using the CLI tools available in the spine-conductor repo.

19.5.2 Creating Git tags and publishing to PyPI

1. Check out the spine-conductor repo, and install it, either in a virtual environment or using `pipx`.
2. You can create a TOML configuration file as mentioned in the README of the repo; say `release.toml`. Something like the sample below should work.

Listing 1: release.toml

```
[tool.conductor]
packagename_regex = "spine(toolbox|(db){0,1}[_-][a-z]+)"  # package name on PyPI

[tool.conductor.dependency_graph]
spinetoolbox = ["spine_items", "spine_engine", "spinedb_api"]
spine_items  = ["spinetoolbox", "spine_engine", "spinedb_api"]
spine_engine = ["spinedb_api"]
spinedb_api  = []

[tool.conductor.repos]
spinetoolbox = "."
spine_items  = "venv/src/spine-items"
spine_engine = "venv/src/spine-engine"
spinedb_api  = "venv/src/spinedb-api"

# # default
# [tool.conductor.branches]
# spinetoolbox = "master"
# spine_items  = "master"
```

(continues on next page)

(continued from previous page)

```
# spine_engine = "master"
# spinedb_api  = "master"
```

3. Now you can create a release by calling the `conduct release -c release.toml` command with the TOML file as config. This starts a guided session where the spine-conductor CLI tool deduces the next version numbers from existing Git tags, updates the corresponding `pyproject.toml` files in all the repos to reflect the new package versions, and finally prompts you to add any edited files, and create the new Git tag. A typical session would like this:

Listing 2: A typical release session; note the JSON summary in the end.

```
$ cd /path/to/repo/Spine-Toolbox
$ conduct release --bump patch -c release.toml  # or include in pyproject.toml
Repository: /path/to/repo/Spine-Toolbox
## master...origin/master
 M pyproject.toml (1)
Select the files to add (comma/space separated list): 1
Creating tag: 0.6.19 @ 034fb4b
Repository: /path/to/repo/venv/src/spine-items
## master...origin/master
 M pyproject.toml (1)
Select the files to add (comma/space separated list): 1
Creating tag: 0.20.1 @ 5848e25
Repository: /path/to/repo/venv/src/spine-engine
## master...origin/master
 M pyproject.toml (1)
Select the files to add (comma/space separated list): 1
Creating tag: 0.22.1 @ e312db2
Repository: /path/to/repo/venv/src/spinedb-api
## master...origin/master
Select the files to add (comma/space separated list):
Creating tag: 0.29.1 @ d9ed86e

Package Tags summary    'pkgtags.json':
{
  "Spine-Toolbox": "0.6.19",
  "spine-items": "0.20.1",
  "spine-engine": "0.22.1",
  "Spine-Database-API": "0.29.1"
}
```

If the session completes successfully, you will see a session summary with the newest Git tags that were created for each package.

4. Push the newly created tags to GitHub. On sh-like shells like: bash, zsh, or git-bash (Windows):

```
for repo in . venv/src/{spinedb-api,spine-{items,engine}}; do
    pushd $repo;
    git push origin master --tags;
    popd
done
```

With Powershell on Windows, something like this should work:

```
"." , "venv/src/spinedb-api", "venv/src/spine-items", "venv/src/spine-engine" | % {
  pushd $_;
  git push origin master --tags;
  popd;
}
```

5. Now you can trigger the workflow to publish the packages to PyPI either by using GitHub CLI, or from the workflow dispatch menu in the spine-conductor repo.

```
cat pkgtags.json | gh workflow run --repo spine-tools/spine-conductor test-n-
↪publish.yml --json
```

   If you are using the workflow dispatch menu, make sure you input the exact same package versions as shown in the summary.

Done! **Note:** Soon, (4) & (5) will be wrapped in a separate command provided by `spine-conductor`.

19.5.3 The `release.toml` file

The config file is a standard TOML file conformant with `pyproject.toml`, meaning all configuration goes under the section `tool.conductor`. The configuration is split into 4 sections: a regex to identify our packages, dependency graph between our packages, path to the repos to be used for the release, and the branches to be used (optional).

1. You can specify a regular expression that will be used to identify "our" packages. Something like the following should work:

Listing 3: Spine package name regular expression

```
[tool.conductor]
packagename_regex = "spine(toolbox|(db){0,1}[_-][a-z]+)"  # package name on PyPI
```

   Note that PyPI treats `-` (hyphen) and `_` (underscore) as equivalent in package names; i.e. `spinedb_api` and `spinedb-api` are equivalent, the regex should accomodate that.

2. The dependency graph between our packages should be specified under the `dependency_graph` section:

Listing 4: Spine package dependency graph

```
[tool.conductor.dependency_graph]
spinetoolbox = ["spine_items", "spine_engine", "spinedb_api"]
spine_items  = ["spinetoolbox", "spine_engine", "spinedb_api"]
spine_engine = ["spinedb_api"]
spinedb_api  = []
```

   Essentially it is a mapping of the "primary" package, and a list of its Spine dependencies.

3. Point to the repository directories *relative* to your current working directory. The following example would be valid if you are preparing the release from the Toolbox repo, and the other Spine package repos are in the virtual environment.

Listing 5: Repository paths

```
[tool.conductor.repos]
spinetoolbox = "."
spine_items  = "venv/src/spine-items"
```

(continues on next page)

(continued from previous page)

```
spine_engine = "venv/src/spine-engine"
spinedb_api  = "venv/src/spinedb-api"
```

4. You can also specify the branches for each repository that should be used for the release. This section is optional, and if left unspecified, the branch name is assumed to be `master`.

Listing 6: Release branches on Spine repositories

```
# default: master
[tool.conductor.branches]
spinetoolbox = "release"
spine_items  = "release"
spine_engine = "release"
spinedb_api  = "release"
```

19.5.4 Manual release (in case of emergency)

This section documents what the spine-conductor CLI tool does under the hood. It is here in case of an emergency (e.g. there's a bug), and the release has to be done manually.

As mentioned earlier, the package version is now derived from Git tags. However, because of the internal dependency between the Spine packages, the versions of the dependencies have to synchronised with the new version. The steps are as follows:

1. Determine the next version for each Spine package. This can be done manually with Git, or you can use `setuptools_scm` in a Python REPL.

   - You can run `git describe --tags` in the repo. This will print out the latest tag followed by a trailer with metadata on distance from the tag; something like this: `0.6.18-100-g411c13e1`. If you want to make a patch release, the next version would be `0.6.19` and a minor release would be `0.7.0`. Repeat this process for all 4 Spine repos.
   - If using a Python REPL, you can do the following for a minor release:

     ```
     >>> from setuptools_scm import get_version
     >>> get_version(".", version_scheme="release-branch-semver")
     '0.7.0.dev100+g411c13e1.d20230823'
     ```

     For a patch release, do the following:

     ```
     >>> get_version(".", version_scheme="guess-next-dev")
     '0.6.19.dev100+g411c13e1.d20230823'
     ```

     Note the first argument to `get_version` is the path to the repository. The above examples assume the repository is your current directory. If it's not, you can provide the path as the first argument.

2. Once the new package versions are determined, you need to edit the `pyproject.toml` files in all 4 repositories with the correct version numbers. For example, in the `Spine-Toolbox` repo if you were to do a minor release, i.e. `0.6.18` → `0.7.0`, the following change would be sufficient:

Listing 7: Example edit to `pyproject.toml` for Spine-Toolbox

```
diff --git a/pyproject.toml b/pyproject.toml
index bd38a2b7..dd9c228e 100644
```

(continues on next page)

(continued from previous page)

```
--- a/pyproject.toml
+++ b/pyproject.toml
@@ -20,8 +20,8 @@ dependencies = [
     "jupyter-client >=6.0",
     "qtconsole >=5.1",
     "sqlalchemy >=1.3",
-    "spinedb_api >=0.29.0",
-    "spine_engine >=0.22.0",
+    "spinedb_api >=0.30.0",
+    "spine_engine >=0.23.0",
     "numpy >=1.20.2",
     "matplotlib >= 3.5",
     "scipy >=1.7.1",
@@ -30,7 +30,7 @@ dependencies = [
     "pygments >=2.8",
     "jill >=0.9.2",
     "pyzmq >=21.0",
-    "spine-items >= 0.20.0",
+    "spine-items >= 0.21.0",
 ]

 [project.urls]
```

3. After updating the `pyproject.toml` file for all 4 Spine repos as above, add and commit the changes in all repos:

```
git commit -i pyproject.toml -m "Release 0.7.0"
```

4. Create a Git tag on the latest commit:

```
git tag 0.7.0 HEAD
```

5. Push the tags to GitHub. On sh-like shells like: bash, zsh, or git-bash (Windows):

Listing 8: Recipe to push Git tags to GitHub on sh-like shells (bash, zsh, git-bash)

```
for repo in . venv/src/{spinedb-api,spine-{items,engine}}; do
    pushd $repo;
    git push origin master --tags;
    popd
done
```

With Powershell on Windows:

Listing 9: Recipe to push Git tags to GitHub on Powershell

```
"." , "venv/src/spinedb-api", "venv/src/spine-items", "venv/src/spine-engine" | % {
  pushd $_;
  git push origin master --tags;
  popd;
}
```

6. Now you can trigger the workflow to publish the packages to PyPI from the workflow dispatch menu in the spine-conductor repo. Ensure you input the exact same package versions as in the tags.

7. In case the workflow above also fails, you have to build the source distribution archive and wheels locally and upload to PyPI manually.

   To build, ensure you have `build` installed. The `build` backend ensures build isolation, and reproducibility of the wheels given a source distribution.

   Listing 10: Build distribution archives and wheels

   ```
   python -m pip install build
   python -m build
   ```

   Once the build completes, you can find the source tarball and the wheel in `dist/`. Now you may upload these files to PyPI.

   It is good practise to first test using TestPyPI before uploading to PyPI, since releases on PyPI are read-only. You want to avoid mistakes.

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

   Upload the distribution using

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

   See Using TestPyPI for more information. To avoid entering your username and password every time, see Keyring support in twine documentation or generate an API key. If everything went smoothly, you are ready to upload the real index. Again, you need to register to PyPI and ask to become a maintainer of the package you want to upload to. Upload the distribution using

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

Done! Now fix the bug that forced you to do the manual release ;)

CHAPTER

TWENTY

API REFERENCE

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

20.1 spinetoolbox

spinetoolbox package.

20.1.1 Subpackages

spinetoolbox.mvcmodels

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

Submodules

spinetoolbox.mvcmodels.array_model

Contains model for the Array editor widget.

Module Contents

Classes

ArrayModel	Model for the Array parameter_value type.

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

Bases: PySide6.QtCore.QAbstractTableModel

Model for the Array parameter_value type.

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

[1] Created with sphinx-autoapi

Parameters
parent (*QObject*) – parent object

array()

Returns the array modeled by this model.

batch_set_data(*indexes*, *values*)

Sets data at multiple indexes at once.

Parameters

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

columnCount(*parent=QModelIndex()*)

Returns 2.

_convert_to_data_type(*indexes*, *values*)

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

Parameters

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

Returns
indexes and converted values

Return type
tuple

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

Returns model's data for given role.

flags(*index*)

Returns table cell's flags.

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

Returns header data.

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

Inserts rows to the array.

is_expanse_row(*row*)

Returns True if row is the expanse row.

Parameters
row (*int*) – a row

Returns
True is row is expanse row, False otherwise

Return type
bool

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

Removes rows from the array.

reset(*value*)

Resets the model to a new array.

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

rowCount(*parent=QModelIndex()*)

Returns the length of the array.

Note: returns 1 even if the array is empty.

set_array_type(*new_type*)

Changes the data type of array's elements.

Parameters
new_type (*Type*) – new element type

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

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

Sets the value at given index.

spinetoolbox.mvcmodels.compound_table_model

Models that vertically concatenate two or more table models.

Module Contents

Classes

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

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

Bases: *spinetoolbox.mvcmodels.minimal_table_model.MinimalTableModel*

A model that concatenates several sub table models vertically.

Initializes model.

Parameters

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

refreshed

map_to_sub(*index*)

Returns an equivalent submodel index.

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

Returns
the equivalent index in one of the submodels

Return type
QModelIndex

`map_from_sub`(*sub_model*, *sub_index*)

Returns an equivalent compound model index.

Parameters

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

Returns
the equivalent index in the compound model

Return type
QModelIndex

`item_at_row`(*row*)

Returns the item at given row.

Parameters
row (*int*) –

Returns
object

`sub_model_at_row`(*row*)

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

Parameters
row (*int*) –

Returns
MinimalTableModel

`sub_model_row`(*row*)

Calculates sub model row.

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

Returns
row in sub model

Return type
int

`refresh`()

Refreshes the layout by computing a new row map.

`_do_refresh`()

Recomputes the row and inverse row maps.

`_append_row_map`(*row_map*)

Appends given row map to the tail of the model.

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

_row_map_iterator_for_model(*model*)

Yields row map for given model. The base class implementation just yields all model rows.

Parameters
model (MinimalTableModel) –

Yields
tuple – (model, row number)

_row_map_for_model(*model*)

Returns row map for given model. The base class implementation just returns all model rows.

Parameters
model (MinimalTableModel) –

Returns
tuples (model, row number)

Return type
list

canFetchMore(*parent*)

Returns True if any of the submodels that haven't been fetched yet can fetch more.

fetchMore(*parent*)

Fetches the next sub model and increments the fetched counter.

flags(*index*)

Return index flags.

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

Returns the data stored under the given role for the item referred to by the index.

Parameters

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

Returns
Item data for given role.

rowCount(*parent=QModelIndex()*)

Returns the sum of rows in all models.

batch_set_data(*indexes*, *data*)

Sets data for indexes in batch. Distributes indexes and values among the different submodels and calls batch_set_data on each of them.

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

Inserts count rows after the given row under the given parent. Localizes the appropriate submodel and calls insertRows on it.

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

Removes count rows starting with the given row under parent. Localizes the appropriate submodels and calls removeRows on it.

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

Bases: *CompoundTableModel*

A compound parameter table model where the last model is an empty row model.

Initializes model.

Parameters

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

property single_models

property empty_model

abstract _create_empty_model()

Creates and returns an empty model.

Returns
model

Return type
EmptyRowModel

init_model()

Initializes the compound model.

Basically populates the sub_models list attribute with the result of _create_empty_model.

_connect_single_model(*model*)

Connects signals so changes in the submodels are acknowledged by the compound.

_recompute_empty_row_map()

Recomputes the part of the row map corresponding to the empty model.

_handle_empty_rows_removed(*parent*, *empty_first*, *empty_last*)

Updates row_map when rows are removed from the empty model.

_handle_empty_rows_inserted(*parent*, *empty_first*, *empty_last*)

Runs when rows are inserted to the empty model. Updates row_map, then emits rowsInserted so the new rows become visible.

_handle_single_model_about_to_be_reset(*model*)

Runs when given model is about to reset.

_handle_single_model_reset(*model*)

Runs when given model is reset.

_refresh_single_model(*model*)

_get_insert_position(*model*)

_insert_single_model(*model*)

_get_row_for_insertion(*pos*)

_insert_row_map(*pos*, *single_row_map*)

clear_model()

Clears the model.

spinetoolbox.mvcmodels.empty_row_model

Contains a table model with an empty last row.

Module Contents

Classes

EmptyRowModel	A table model with a last empty row.

class spinetoolbox.mvcmodels.empty_row_model.**EmptyRowModel**(*parent=None*, *header=None*)

Bases: *spinetoolbox.mvcmodels.minimal_table_model.MinimalTableModel*

A table model with a last empty row.

Init class.

canFetchMore(*_parent*)

Return True if the model hasn't been fetched.

fetchMore(*parent*)

Fetch data and use it to reset the model.

flags(*index*)

Return default flags except if forcing defaults.

set_default_row(***kwargs*)

Set default row data.

clear()

Clear all data in model.

reset_model(*main_data=None*)

Reset model.

_handle_data_changed(*top_left*, *bottom_right*, *roles=None*)

Insert a new last empty row in case the previous one has been filled with any data other than the defaults.

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

Don't remove the last empty row.

_handle_rows_inserted(*parent*, *first*, *last*)

Handle rowsInserted signal.

set_rows_to_default(*first*, *last=None*)

Set default data in newly inserted rows.

spinetoolbox.mvcmodels.file_list_models

Contains a generic File list model and an Item for that model.

Module Contents

Classes

FileListModel	A model for files to be shown in a file tree view.
CommandLineArgItem	
NewCommandLineArgItem	
CommandLineArgsModel	
JumpCommandLineArgsModel	

class spinetoolbox.mvcmodels.file_list_models.**FileListModel**(*header_label=''*, *draggable=False*)

Bases: PySide6.QtCore.QAbstractItemModel

A model for files to be shown in a file tree view.

Parameters

- **header_label** (*str*) – header label
- **draggable** (*bool*) – if True, the top level items are drag and droppable

FileItem

PackItem

rowCount(*parent=QModelIndex()*)

columnCount(*parent=QModelIndex()*)

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

Returns header information.

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

Returns data associated with given role at given index.

flags(*index*)

mimeData(*indexes*)

resource(*index*)

Returns the resource at given index.

Parameters
index (*QModelIndex*) – index

Returns
resource

Return type
ProjectItemResource

parent(*index*)

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

update(*resources*)

Updates the model according to given list of resources.

Parameters
resources (*Iterable of ProjectItemResource*) – resources

duplicate_paths()

Checks if resources in the model have duplicate file paths.

Returns
set of duplicate file paths

Return type
set of str

_pack_index(*pack_label*)

Finds a pack's index in pack resources list.

Parameters
pack_label (*str*) – pack label

Returns
index to pack resources list

Return type
int

class spinetoolbox.mvcmodels.file_list_models.**CommandLineArgItem**(*text=''*, *rank=None*, *selectable=False*, *editable=False*, *drag_enabled=False*, *drop_enabled=False*)

Bases: PySide6.QtGui.QStandardItem

set_rank(*rank*)

static _make_icon(*rank=None*)

setData(*value*, *role=Qt.ItemDataRole.UserRole + 1*)

class spinetoolbox.mvcmodels.file_list_models.**NewCommandLineArgItem**

Bases: *CommandLineArgItem*

setData(*value*, *role=Qt.ItemDataRole.UserRole + 1*)

class spinetoolbox.mvcmodels.file_list_models.**CommandLineArgsModel**(*parent=None*)

Bases: PySide6.QtGui.QStandardItemModel

property args

args_updated

append_arg(*arg*)

replace_arg(*row*, *arg*)

mimeData(*indexes*)

dropMimeData(*data*, *drop_action*, *row*, *column*, *parent*)

static _reset_root(*root*, *args*, *child_params*, *has_empty_row=True*)

class spinetoolbox.mvcmodels.file_list_models.**JumpCommandLineArgsModel**(*parent=None*)

Bases: *CommandLineArgsModel*

reset_model(*args*)

canDropMimeData(*data*, *drop_action*, *row*, *column*, *parent*)

spinetoolbox.mvcmodels.filter_checkbox_list_model

Provides FilterCheckboxListModel for FilterWidget.

Module Contents

Classes

SimpleFilterCheckboxListModel	**param parent** parent widget
LazyFilterCheckboxListModel	Extends SimpleFilterCheckboxListModel to allow for lazy loading in synch with another model.
DataToValueFilterCheckboxListModel	Extends SimpleFilterCheckboxListModel to allow for translating internal data to a value for display role.

class spinetoolbox.mvcmodels.filter_checkbox_list_model.**SimpleFilterCheckboxListModel**(*parent*, *show_empty=True*)

Bases: PySide6.QtCore.QAbstractListModel

Parameters

- **parent** (*QWidget*) – parent widget
- **show_empty** (*bool*) – if True, adds an empty row to the end of the list

property _show_empty

property _show_add_to_selection

_SELECT_ALL_STR = '(Select all)'

_SELECT_ALL_FILTERED_STR = '(Select all filtered)'

_EMPTY_STR = '(Empty)'

_ADD_TO_SELECTION_STR = 'Add current selection to filter'

reset_selection()

_handle_select_all_clicked()

_check_all_selected()

rowCount(*parent=QModelIndex()*)

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

_handle_index_clicked(*index*)

set_list(*data*, *all_selected=True*)

filter_by_condition(*condition*)

Updates selected items by applying a condition.

Parameters
condition (*function*) – Filter acceptance condition.

set_selected(*selected*, *select_empty=None*)

get_selected()

get_not_selected()

set_filter(*filter_expression*)

search_filter_expression(*item*)

apply_filter()

_remove_and_add_filtered()

_remove_and_replace_filtered()

remove_filter()

_do_add_items(*data*)

add_items(*data*, *selected=None*)

remove_items(*data*)

class spinetoolbox.mvcmodels.filter_checkbox_list_model.**LazyFilterCheckboxListModel**(*parent*, *db_mngr*, *db_maps*, *fetch_parent*, *show_empty=True*)

Bases: *SimpleFilterCheckboxListModel*

Extends SimpleFilterCheckboxListModel to allow for lazy loading in synch with another model.

Parameters

- **parent** (SpineDBEditor) – parent widget
- **db_mngr** (SpineDBManager) – database manager
- **db_maps** (*Sequence of DatabaseMapping*) – database maps
- **fetch_parent** (FetchParent) – fetch parent

- **show_empty** (*bool*) – if True, show an empty row at the end of the list

canFetchMore(*_parent*)

fetchMore(*_parent*)

_do_add_items(*data*)

Adds items so the list is always sorted, while assuming that both existing and new items are sorted.

class spinetoolbox.mvcmodels.filter_checkbox_list_model.**DataToValueFilterCheckboxListModel**(*parent*, *data_to_value*, *show_empty=Tr*

Bases: *SimpleFilterCheckboxListModel*

Extends SimpleFilterCheckboxListModel to allow for translating internal data to a value for display role.

Parameters

- **parent** (SpineDBEditor) – parent widget
- **data_to_value** (*method*) – a method to translate item data to a value for display role
- **show_empty** (*bool*) – if True, add an empty row to the end of the list

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

search_filter_expression(*item*)

spinetoolbox.mvcmodels.filter_execution_model

Contains FilterExecutionModel.

Module Contents

Classes

FilterExecutionModel

class spinetoolbox.mvcmodels.filter_execution_model.**FilterExecutionModel**

Bases: PySide6.QtCore.QAbstractListModel

_filter_consoles

reset_model(*filter_consoles*)

rowCount(*parent=QModelIndex()*)

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

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

find_index(*console_key*)

get_console(*filter_id*)

spinetoolbox.mvcmodels.indexed_value_table_model

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

Module Contents

Classes

IndexedValueTableModel	A base class for time pattern and time series models.

Attributes

EXPANSE_COLOR

spinetoolbox.mvcmodels.indexed_value_table_model.**EXPANSE_COLOR**

class spinetoolbox.mvcmodels.indexed_value_table_model.**IndexedValueTableModel**(*value*, *parent*)

Bases: PySide6.QtCore.QAbstractTableModel

A base class for time pattern and time series models.

Parameters

- **value** (*IndexedValue*) – a parameter_value
- **parent** (*QObject*) – parent object

property value

Returns the parameter_value associated with the model.

columnCount(*parent=QModelIndex()*)

Returns the number of columns which is two.

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

Returns the data at index for given role.

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

Returns a header.

is_expanse_row(*row*)

Returns True if row is the expanse row.

Parameters
row (*int*) – a row

Returns
True if row is the expanse row, False otherwise

Return type
bool

reset(*value*)

Resets the model.

rowCount(*parent=QModelIndex()*)

Returns the number of rows.

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

spinetoolbox.mvcmodels.map_model

A model for maps, used by the parameter_value editors.

Module Contents

Classes

MapModel	A model for Map type parameter values.

Functions

_rows_to_dict(rows)	Turns table into nested dictionaries.
_reconstruct_map(tree)	Constructs a `Map` from a nested dictionary.
_data_length(row)	Counts the number of non-empty elements at the beginning of row.
_gather_index_names(map_value)	Collects index names from Map.
_apply_index_names(map_value, index_names)	Applies index names to Map.
_numpy_string_to_python_strings(rows)	Converts instances of `numpy.str_` to regular Python strings.

Attributes

empty	Sentinel for empty cells.

spinetoolbox.mvcmodels.map_model.**empty**

Sentinel for empty cells.

class spinetoolbox.mvcmodels.map_model.**MapModel**(*map_value*, *parent*)

Bases: `PySide6.QtCore.QAbstractTableModel`

A model for Map type parameter values.

This model represents the Map as a 2D table. Each row consists of one or more index columns and a value column. The last columns of a row are padded with Nones.

Example

```
Map {
    "A": 1.0
    "B": Map {"a": -1.0}
    "C": 3.0
}
```

The table corresponding to the above map:

“A”	1.0	None
“B”	“a”	-1.0
“C”	3.0	None

Parameters

- **map_value** (*Map*) – a map
- **parent** (*QObject*) – parent object

append_column()

Appends a new column to the right.

clear(*indexes*)

Clears table cells.

Parameters
indexes (*list of QModelIndex*) – indexes to clear

columnCount(*index=QModelIndex()*)

Returns the number of columns in this model.

convert_leaf_maps()

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

Returns the data associated with the given role.

flags(*index*)

Returns flags at index.

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

Returns row numbers for vertical headers and column titles for horizontal ones.

insertColumns(*column*, *count*, *parent=QModelIndex()*)

Inserts new columns into the map.

Parameters

- **column** (*int*) – column index where to insert
- **count** (*int*) – number of new columns
- **parent** (*QModelIndex*) – ignored

Returns
True if insertion was successful, False otherwise

Return type
bool

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

Inserts new rows into the map.

Parameters

- **row** (*int*) – an index where to insert the new data
- **count** (*int*) – number of rows to insert
- **parent** (*QModelIndex*) – an index to a parent model

Returns
True if the operation was successful

Return type
bool

is_leaf_value(*index*)

Checks if given model index contains a leaf value.

Parameters
index (*QModelIndex*) – index to check

Returns
True if index points to leaf value, False otherwise

Return type
bool

_is_in_expanse(*row*, *column*)

Returns True, if given row and column is in the right or bottom 'expanding' zone.

Parameters

- **row** (*int*) – row index
- **column** (*int*) – column index

Returns
True if the cell is in the expanse, False otherwise

Return type
bool

is_expanse_column(*column*)

Returns True if given column is the expanse column.

Parameters
column (*int*) – column

Returns
True if column is expanse column, False otherwise

Return type
bool

is_expanse_row(*row*)

Returns True if given row is the expanse row.

Parameters
row (*int*) – row

Returns
True if row is the expanse row, False otherwise

Return type
bool

removeColumns(*column*, *count*, *parent=QModelIndex()*)

Removes columns from the map.

Parameters

- **column** (*int*) – first column to remove
- **count** (*int*) – number of columns to remove
- **parent** (*QModelIndex*) – an index to a parent model

Returns
True if the operation was successful

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

Removes rows from the map.

Parameters

- **row** (*int*) – first row to remove
- **count** (*int*) – number of rows to remove
- **parent** (*QModelIndex*) – an index to a parent model

Returns
True if the operation was successful

reset(*map_value*)

Resets the model to given map_value.

rowCount(*parent=QModelIndex()*)

Returns the number of rows.

set_box(*top_left*, *bottom_right*, *data*)

Sets data for several indexes at once.

Parameters

- **top_left** (*QModelIndex*) – a sequence of model indexes
- **bottom_right** (*QModelIndex*) – a sequence of values corresponding to the indexes
- **data** (*list of list*) – box of data

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

Sets data in the map.

Parameters

- **index** (*QModelIndex*) – an index to the model
- **value** (*object*) – JSON representation of the value
- **role** (*int*) – a role

Returns
True if the operation was successful

Return type
bool

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

trim_columns()

Removes empty columns from the right.

value()

Returns the Map.

index_name(*index*)

spinetoolbox.mvcmodels.map_model.**_rows_to_dict**(*rows*)

Turns table into nested dictionaries.

Parameters
rows (*list*) – a list of row data

Returns
a nested dictionary

Return type
dict

spinetoolbox.mvcmodels.map_model.**_reconstruct_map**(*tree*)

Constructs a Map from a nested dictionary.

Parameters
tree (*dict*) – a nested dictionary

Returns
reconstructed Map

Return type
Map

spinetoolbox.mvcmodels.map_model.**_data_length**(*row*)

Counts the number of non-empty elements at the beginning of row.

Parameters
row (*list*) – a row of data

Returns
data length

Return type
int

spinetoolbox.mvcmodels.map_model.**_gather_index_names**(*map_value*)

Collects index names from Map.

Returns only the 'first' index name for nested maps at the same depth.

Parameters
map_value (*Map*) – map to investigate

Returns
index names

Return type
list of str

spinetoolbox.mvcmodels.map_model.**_apply_index_names**(*map_value*, *index_names*)

Applies index names to Map.

Parameters

- **map_value** (*Map*) – target Map
- **index_names** (*list of str*) – index names

spinetoolbox.mvcmodels.map_model.**_numpy_string_to_python_strings**(*rows*)

Converts instances of `numpy.str_` to regular Python strings.

Parameters
rows (*list of list*) – table rows

Returns
converted rows

Return type
list of list

spinetoolbox.mvcmodels.minimal_table_model

Contains a minimal table model.

Module Contents

Classes

MinimalTableModel	Table model for outlining simple tabular data.

class spinetoolbox.mvcmodels.minimal_table_model.**MinimalTableModel**(*parent=None*, *header=None*, *lazy=True*)

Bases: `PySide6.QtCore.QAbstractTableModel`

Table model for outlining simple tabular data.

Parameters

- **parent** (*QObject, optional*) – the parent object
- **header** (*list of str*) – header labels
- **lazy** (*boolean*) – if True, fetches data lazily

clear()

Clear all data in model.

flags(*index*)

Return index flags.

canFetchMore(*parent*)

Return True if the model hasn't been fetched.

fetchMore(*parent*)

Fetch data and use it to reset the model.

rowCount(*parent=QModelIndex()*)

Number of rows in the model.

columnCount(*parent=QModelIndex()*)

Number of columns in the model.

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

Returns headers.

set_horizontal_header_labels(*labels*)

Set horizontal header labels.

insert_horizontal_header_labels(*section*, *labels*)

Insert horizontal header labels at the given section.

horizontal_header_labels()

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

Sets the data for the given role and section in the header with the specified orientation to the value supplied.

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

Returns the data stored under the given role for the item referred to by the index.

Parameters

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

Returns

Item data for given role.

row_data(*row*, *role=Qt.ItemDataRole.DisplayRole*)

Returns the data stored under the given role for the given row.

Parameters

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

Returns

Row data for given role.

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

Set data in model.

batch_set_data(*indexes*, *data*)

Batch set data for indexes.

Parameters

- **indexes** (*Iterable of QModelIndex*) – model indexes
- **data** (*Iterable*) – data at each index

Returns

True if data was set successfully, False otherwise

Return type

boolean

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

Inserts count rows into the model before the given row. Items in the new row will be children of the item represented by the parent model index.

Parameters

- **row** (*int*) – Row number where new rows are inserted
- **count** (*int*) – Number of inserted rows
- **parent** (*QModelIndex*) – Parent index

Returns

True if rows were inserted successfully, False otherwise

insertColumns(*column*, *count*, *parent=QModelIndex()*)

Inserts count columns into the model before the given column. Items in the new column will be children of the item represented by the parent model index.

Parameters

- **column** (*int*) – Column number where new columns are inserted
- **count** (*int*) – Number of inserted columns
- **parent** (*QModelIndex*) – Parent index

Returns

True if columns were inserted successfully, False otherwise

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

Removes count rows starting with the given row under parent.

Parameters

- **row** (*int*) – Row number where to start removing rows
- **count** (*int*) – Number of removed rows
- **parent** (*QModelIndex*) – Parent index

Returns

True if rows were removed successfully, False otherwise

removeColumns(*column*, *count*, *parent=QModelIndex()*)

Removes count columns starting with the given column under parent.

Parameters

- **column** (*int*) – Column number where to start removing columns
- **count** (*int*) – Number of removed columns
- **parent** (*QModelIndex*) – Parent index

Returns

True if columns were removed successfully, False otherwise

reset_model(*main_data=None*)

Reset model.

spinetoolbox.mvcmodels.minimal_tree_model

Models to represent items in a tree.

Module Contents

Classes

TreeItem	A tree item that can fetch its children.
MinimalTreeModel	Base class for all tree models.

class spinetoolbox.mvcmodels.minimal_tree_model.**TreeItem**(*model*)

A tree item that can fetch its children.

Parameters
model (MinimalTreeModel) – The model where the item belongs.

property model

property children

property parent_item

property display_data

property edit_data

set_has_children_initially(*has_children_initially*)

has_children()

Returns whether this item has or could have children.

is_valid()

Tests if item is valid.

Returns
True if item is valid, False otherwise

Return type
bool

child(*row*)

Returns the child at given row or None if out of bounds.

last_child()

Returns the last child.

child_count()

Returns the number of children.

row_count()

Returns the number of rows, which may be different from the number of children. This allows subclasses to hide children.

child_number()

Returns the rank of this item within its parent or -1 if it's an orphan.

find_children(*cond=lambda child: ...*)

Returns children that meet condition expressed as a lambda function.

find_child(*cond=lambda child: ...*)

Returns first child that meet condition expressed as a lambda function or None.

next_sibling()

Returns the next sibling or None if it's the last.

previous_sibling()

Returns the previous sibling or None if it's the first.

index()

set_up()

_do_set_up()

Do stuff after the item has been inserted.

_polish_children(*children*)

Polishes children just before inserting them.

insert_children(*position*, *children*)

Insert new children at given position. Returns a boolean depending on how it went.

Parameters

- **position** (*int*) – insert new items here
- **children** (*list of TreeItem*) – insert items from this iterable

Returns

True if the children were inserted successfully, False otherwise

Return type

bool

append_children(*children*)

Append children at the end.

tear_down()

Do stuff after the item has been removed.

tear_down_recursively()

remove_children(*position*, *count*)

Removes count children starting from the given position.

Parameters

- **position** (*int*) – position of the first child to remove
- **count** (*int*) – number of children to remove

Returns

True if operation was successful, False otherwise

Return type

bool

flags(*column*)

Enables the item and makes it selectable.

data(*column*, *role=Qt.ItemDataRole.DisplayRole*)

Returns data for given column and role.

can_fetch_more()

Returns whether this item can fetch more.

fetch_more()

Fetches more children.

abstract set_data(*column*, *value*, *role*)

Sets data for this item.

Parameters

- **column** (*int*) – column index
- **value** (*object*) – a new value
- **role** (*int*) – role of the new value

Returns
True if data was set successfully, False otherwise

Return type
bool

class spinetoolbox.mvcmodels.minimal_tree_model.**MinimalTreeModel**(*parent*)

Bases: PySide6.QtCore.QAbstractItemModel

Base class for all tree models.

Init class.

Parameters
parent (SpineDBEditor) –

visit_all(*index=QModelIndex()*, *view=None*)

Iterates all items in the model including and below the given index. Iterative implementation so we don't need to worry about Python recursion limits.

Parameters

- **index** (*QModelIndex*) – an index to start. If not given, we start at the root
- **view** (*QTreeView*) – a tree view. If given, we only yield items that are visible to that view. So for example, if a tree item is not expanded then we don't yield its children.

Yields
TreeItem

item_from_index(*index*)

Return the item corresponding to the given index.

Parameters
index (*QModelIndex*) – model index

Returns
item at index

Return type
TreeItem

index_from_item(*item*)

Return a model index corresponding to the given item.

Parameters
item (StandardTreeItem) – item

Returns
item's index

Return type
QModelIndex

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

Returns the index of the item in the model specified by the given row, column and parent index.

parent(*index*)

Returns the parent of the model item with the given index.

columnCount(*parent=QModelIndex()*)

rowCount(*parent=QModelIndex()*)

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

Returns the data stored under the given role for the index.

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

Sets data for given index and role. Returns True if successful; otherwise returns False.

flags(*index*)

Returns the item flags for the given index.

hasChildren(*parent*)

canFetchMore(*parent*)

fetchMore(*parent*)

spinetoolbox.mvcmodels.project_item_specification_models

Contains a class for storing Tool specifications.

Module Contents

Classes

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

class spinetoolbox.mvcmodels.project_item_specification_models.**ProjectItemSpecificationModel**(*icons*)

Bases: PySide6.QtCore.QAbstractListModel

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

specification_replaced

add_specification(*name*)

Adds a specification to the model.

Parameters
name (*str*) – specification's name

remove_specification(*name*)

Removes a specification from the model

Parameters
name (*str*) – specification's name

replace_specification(*old_name*, *new_name*)

Replaces a specification.

Parameters

- **old_name** (*str*) – previous name
- **new_name** (*str*) – new name

connect_to_project(*project*)

Connects the model to a project.

Parameters
project (SpineToolboxProject) – project to connect to

clear()

rowCount(*parent=None*)

Returns the number of specs in the model.

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

Returns
Number of rows (available specs) in the model

data(*index*, *role=None*)

Must be reimplemented when subclassing.

Parameters

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

Returns
Data according to requested role

flags(*index*)

Returns enabled flags for the given index.

Parameters
index (*QModelIndex*) – Index of spec

insertRow(*spec_name*, *row=None*, *parent=QModelIndex()*)

Insert row (specification) into model.

Parameters

- **spec_name** (*str*) – name of spec added to the model

- **row** (*int, optional*) – Row to insert spec to
- **parent** (*QModelIndex*) – Parent of child (not used)

Returns
Void

removeRow(*row*, *parent=QModelIndex()*)

Remove row (spec) from model.

Parameters

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

Returns
Boolean variable

specification(*row*)

Returns spec on given row.

Parameters
row (*int*) – Row of spec specification

Returns
ProjectItemSpecification from specification list or None if given row is zero

specification_row(*name*)

Returns the row on which the given specification is located or -1 if it is not found.

specification_index(*name*)

Returns the QModelIndex on which a specification with the given name is located or invalid index if it is not found.

class spinetoolbox.mvcmodels.project_item_specification_models.**FilteredSpecificationModel**(*item_type*)

Bases: PySide6.QtCore.QSortFilterProxyModel

filterAcceptsRow(*source_row*, *source_parent*)

get_mime_data_text(*index*)

specifications()

Yields all specs.

specification(*row*)

spinetoolbox.mvcmodels.resource_filter_model

Contains ResourceFilterModel.

Module Contents

Classes

ResourceFilterModel	**param connection** connection whose resources to model

class spinetoolbox.mvcmodels.resource_filter_model.**ResourceFilterModel**(*connection*, *project*, *undo_stack*, *logger*)

Bases: PySide6.QtGui.QStandardItemModel

Parameters

- **connection** (LoggingConnection) – connection whose resources to model
- **project** (SpineToolboxProject) – project
- **undo_stack** (*QUndoStack*) – an undo stack
- **logger** (LoggerInterface) – a logger

property connection

tree_built

_SELECT_ALL = 'Select all'

FILTER_TYPES

FILTER_TYPE_TO_TEXT

build_tree()

Rebuilds model's contents.

fetch_filters()

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

_change_filter_checked_state(*index*, *is_on*)

Changes the online status of the filter item at index.

Parameters

- **index** (*QModelIndex*) – item's index
- **is_on** (*bool*) – True if filter are turned online, False otherwise

set_online(*resource*, *filter_type*, *online*)

Sets the given filters online or offline.

Parameters

- **resource** (*str*) – Resource label
- **filter_type** (*str*) – Always SCENARIO_FILTER_TYPE, for now.
- **online** (*dict*) – mapping from scenario/tool id to online flag

_find_filter_type_item(*resource*, *filter_type*)

Searches for filter type item.

Parameters

- **resource** (*str*) – resource label
- **filter_type** (*str*) – filter type identifier

Returns
filter type item or None if not found

Return type
QStandardItem

filter_type_items(*filter_type*)

An iterator to filter type items.

Parameters
filter_type (*str*) – filter type

Yields
QStandardItem – filter type item

_set_all_selected_item(*resource*, *filter_type_item*, *emit_data_changed=False*)

Updates 'Select All' item's checked state.

Parameters

- **resource** (*str*) – resource label
- **filter_type_item** (*QStandardItem*) – filter type item
- **emit_data_changed** (*bool*) – if True, emit dataChanged signal if the state was updated

set_filter_type_enabled(*filter_type*, *enabled*)

Enables or disables a filter type.

Parameters

- **filter_type** (*str*) – filter type
- **enabled** (*bool*) – whether the filter is enabled

spinetoolbox.mvcmodels.shared

Contains stuff that is used by more than one model.

Module Contents

spinetoolbox.mvcmodels.shared.**PARSED_ROLE**

spinetoolbox.mvcmodels.shared.**DB_MAP_ROLE**

spinetoolbox.mvcmodels.time_pattern_model

A model for time patterns, used by the parameter_value editors.

Module Contents

Classes

TimePatternModel	A model for time pattern type parameter values.

class spinetoolbox.mvcmodels.time_pattern_model.**TimePatternModel**(*value*, *parent*)

Bases: *spinetoolbox.mvcmodels.indexed_value_table_model.IndexedValueTableModel*

A model for time pattern type parameter values.

Parameters

- **value** (*IndexedValue*) – a parameter_value
- **parent** (*QObject*) – parent object

flags(*index*)

Returns flags at index.

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

Inserts new time period - value pairs into the pattern.

New time periods are initialized to empty strings and the corresponding values to zeros.

Parameters

- **row** (*int*) – an index where to insert the new data
- **count** (*int*) – number of time period - value pairs to insert
- **parent** (*QModelIndex*) – an index to a parent model

Returns

True if the operation was successful

Return type

bool

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

Removes time period - value pairs from the pattern.

Parameters

- **row** (*int*) – an index where to remove the data
- **count** (*int*) – number of time period - value pairs to remove
- **parent** (*QModelIndex*) – an index to a parent model

Returns

True if the operation was successful

Return type

bool

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

Sets a time period or a value in the pattern.

Column index 0 corresponds to the time periods while 1 corresponds to the values.

Parameters

- **index** (*QModelIndex*) – an index to the model
- **value** (*str, float*) – a new time period or value
- **role** (*int*) – a role

Returns
True if the operation was successful

Return type
bool

batch_set_data(*indexes*, *values*)

Sets data for several indexes at once.

Parameters

- **indexes** (*Sequence*) – a sequence of model indexes
- **values** (*Sequence*) – a sequence of time periods/floats corresponding to the indexes

spinetoolbox.mvcmodels.time_series_model_fixed_resolution

A model for fixed resolution time series, used by the parameter_value editors.

Module Contents

Classes

TimeSeriesModelFixedResolution	A model for fixed resolution time series type parameter values.

class spinetoolbox.mvcmodels.time_series_model_fixed_resolution.**TimeSeriesModelFixedResolution**(*series*, *parent*)

Bases: *spinetoolbox.mvcmodels.indexed_value_table_model.IndexedValueTableModel*

A model for fixed resolution time series type parameter values.

Parameters

- **series** (*TimeSeriesFixedResolution*) – a time series
- **parent** (*QObject*) – parent object

property indexes

Returns the time stamps as an array.

property values

Returns the values of the time series as an array.

flags(*index*)

Returns flags at index.

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

Inserts new values to the series.

The new values are set to zero. Start time or resolution are left unchanged.

Parameters

- **row** (*int*) – a numeric index to the first stamp/value to insert
- **count** (*int*) – number of stamps/values to insert
- **parent** (*QModelIndex*) – index to a parent model

Returns

True if the operation was successful

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

Removes values from the series.

Parameters

- **row** (*int*) – a numeric index to the series where to begin removing
- **count** (*int*) – how many stamps/values to remove
- **parent** (*QModelIndex*) – an index to the parent model

Returns

True if the operation was successful.

reset(*value*)

Resets the model with new time series data.

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

Sets a given value in the series.

Column index 1 refers to values. Note it does not make sense to set the time stamps in fixed resolution series.

Parameters

- **index** (*QModelIndex*) – an index to the model
- **value** (*numpy.datetime64, float*) – a new stamp or value
- **role** (*int*) – a role

Returns

True if the operation was successful

batch_set_data(*indexes*, *values*)

Sets data for several indexes at once.

Only the values of the series are modified as the time stamps are immutable.

Parameters

- **indexes** (*Sequence*) – a sequence of model indexes
- **values** (*Sequence*) – a sequence of floats corresponding to the indexes

set_ignore_year(*ignore_year*)

Sets the ignore_year option of the time series.

set_repeat(*repeat*)

Sets the repeat option of the time series.

set_resolution(*resolution*)

Sets the resolution.

set_start(*start*)

Sets the start datetime.

spinetoolbox.mvcmodels.time_series_model_variable_resolution

A model for variable resolution time series, used by the parameter_value editors.

Module Contents

Classes

TimeSeriesModelVariableResolution	A model for variable resolution time series type parameter values.

class spinetoolbox.mvcmodels.time_series_model_variable_resolution.**TimeSeriesModelVariableResolution**(*val*
pa
ent

Bases: *spinetoolbox.mvcmodels.indexed_value_table_model.IndexedValueTableModel*

A model for variable resolution time series type parameter values.

Parameters

- **value** (*IndexedValue*) – a parameter_value
- **parent** (*QObject*) – parent object

property indexes

Returns the time stamps as an array.

property values

Returns the values of the time series as an array.

flags(*index*)

Returns the flags for given model index.

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

Inserts new time stamps and values to the series.

When inserting in the middle of the series the new time stamps are distributed evenly among the time span between the two time stamps around the insertion point. When inserting at the beginning or at the end of the series the duration between the new time stamps is set equal to the first/last duration in the original series.

The new values are set to zero.

Parameters

- **row** (*int*) – a numeric index to the first stamp/value to insert
- **count** (*int*) – number of stamps/values to insert
- **parent** (*QModelIndex*) – index to a parent model

Returns
True if the insertion was successful

Return type
bool

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

Removes time stamps/values from the series.

Parameters

- **row** (*int*) – a numeric index to the series where to begin removing
- **count** (*int*) – how many stamps/values to remove
- **parent** (*QModelIndex*) – an index to the parent model

Returns
True if the operation was successful.

Return type
bool

reset(*value*)

Resets the model with new time series data.

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

Sets a given time stamp or value in the series.

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

Parameters

- **index** (*QModelIndex*) – an index to the model
- **value** (*numpy.datetime64, float*) – a new stamp or value
- **role** (*int*) – a role

Returns
True if the operation was successful

Return type
bool

batch_set_data(*indexes*, *values*)

Sets data for several indexes at once.

Parameters

- **indexes** (*Sequence*) – a sequence of model indexes
- **values** (*Sequence*) – a sequence of datetimes/floats corresponding to the indexes

set_ignore_year(*ignore_year*)

Sets the ignore_year option of the time series.

set_repeat(*repeat*)

Sets the repeat option of the time series.

spinetoolbox.project_item

This subpackage contains base classes for project items.

Submodules

spinetoolbox.project_item.logging_connection

Contains logging connection and jump classes.

Module Contents

Classes

HeadlessConnection	A project item connection that is compatible with headless mode.
LoggingConnection	A project item connection that is compatible with headless mode.
LoggingJump	Represents a conditional jump between two project items.

Attributes

_DATABASE_ITEM_TYPE

spinetoolbox.project_item.logging_connection.**_DATABASE_ITEM_TYPE**

class spinetoolbox.project_item.logging_connection.**HeadlessConnection**(*source_name*, *source_position*, *destination_name*, *destination_position*, *options=None*, *filter_settings=None*, *legacy_resource_filter_ids=None*)

Bases: spine_engine.project_item.connection.ResourceConvertingConnection

A project item connection that is compatible with headless mode.

Parameters

- **source_name** (*str*) – source project item's name
- **source_position** (*str*) – source anchor's position
- **destination_name** (*str*) – destination project item's name

- **destination_position** (*str*) – destination anchor's position
- **options** (*dict, optional*) – any additional options
- **filter_settings** (*FilterSettings, optional*) – filter settings

property database_resources

Connection's database resources

set_filter_enabled(*resource_label*, *filter_type*, *filter_name*, *enabled*)

Enables or disables a filter.

Parameters

- **resource_label** (*str*) – database resource name
- **filter_type** (*str*) – filter type
- **filter_name** (*str*) – filter name
- **enabled** (*bool*) – True to enable the filter, False to disable it

set_filter_type_enabled(*filter_type*, *enabled*)

Enables or disables a filter type.

Parameters

- **filter_type** (*str*) – filter type
- **enabled** (*bool*) – True to enable the filter type, False to disable it

_convert_legacy_resource_filter_ids_to_filter_settings()

Converts legacy resource filter ids to filter settings.

This method should be called once after constructing the connection from potentially legacy dict using `from_dict()`.

static _constructor_args_from_dict(*connection_dict*)

See base class.

classmethod from_dict(*connection_dict*, ***kwargs*)

Deserializes a connection from dict.

Parameters

- **connection_dict** (*dict*) – serialized LoggingConnection
- ****kwargs** – additional keyword arguments to be forwarded to class constructor

receive_resources_from_source(*resources*)

See base class.

replace_resources_from_source(*old*, *new*)

Replaces existing resources by new ones.

Parameters

- **old** (*list of ProjectItemResource*) – old resources
- **new** (*list of ProjectItemResource*) – new resources

class spinetoolbox.project_item.logging_connection.**LoggingConnection**(**args*, *toolbox*, ***kwargs*)

Bases: *spinetoolbox.log_mixin.LogMixin*, *HeadlessConnection*

A project item connection that is compatible with headless mode.

Parameters

- **source_name** (*str*) – source project item's name
- **source_position** (*str*) – source anchor's position
- **destination_name** (*str*) – destination project item's name
- **destination_position** (*str*) – destination anchor's position
- **options** (*dict, optional*) – any additional options
- **filter_settings** (*FilterSettings, optional*) – filter settings

property graphics_item

__hash__()

Return hash(self).

static item_type()

has_filters()

Returns True if connection has any filters.

Returns
True if connection has filters, False otherwise

Return type
bool

_get_db_map(*url*, *ignore_version_error=False*)

_pop_unused_db_maps()

Removes unused database maps and unregisters from listening the DB manager.

_make_fetch_parent(*db_map*, *item_type*)

_fetch_more_if_possible()

_receive_data_changed()

receive_session_committed(*db_maps*, *cookie*)

receive_session_rolled_back(*db_map*)

receive_error_msg(*_db_map_error_log*)

get_filter_item_names(*filter_type*, *url*)

_do_purge_before_writing(*resources*)

may_have_filters()

Returns whether this connection may have filters.

Returns
True if it is possible for the connection to have filters, False otherwise

Return type
bool

`may_have_write_index`()

Returns whether this connection may have write index.

Returns
True if it is possible for the connection to have write index, False otherwise

Return type
bool

`may_use_memory_db`()

Returns whether this connection may use memory DB.

Returns
True if it is possible for the connection to use memory DB, False otherwise

Return type
bool

`may_use_datapackage`()

Returns whether this connection may use datapackage.

Returns
True if it is possible for the connection to use datapackage, False otherwise

Return type
bool

`may_purge_before_writing`()

Returns whether this connection may purge before writing.

Returns
True if it is possible for the connection to purge before writing, False otherwise

Return type
bool

`online_filters`(*resource_label*, *filter_type*)

Returns filter online states for given resource and filter type.

Parameters

- **resource_label** (*str*) – resource label
- **filter_type** (*str*) – filter type

Returns
mapping from filter names to online states

Return type
dict

`set_online`(*resource*, *filter_type*, *online*)

Sets the given filters online or offline.

Parameters

- **resource** (*str*) – Resource label
- **filter_type** (*str*) – filter type
- **online** (*dict*) – mapping from scenario name to online flag

set_filter_default_online_status(*auto_online*)

Sets the auto_online flag.

Parameters
auto_online (*bool*) – If True, unknown filters are online by default

refresh_resource_filter_model()

Makes resource filter mode fetch filter data from database.

set_filter_type_enabled(*filter_type*, *enabled*)

See base class.

receive_resources_from_source(*resources*)

See base class.

replace_resources_from_source(*old*, *new*)

See base class.

set_connection_options(*options*)

Overwrites connections options.

Parameters
options (*dict*) – new options

_check_available_filters()

Cross-checks filter settings with source databases.

Returns
filter settings containing only filters that exist in source databases

Return type
FilterSettings

_resource_filters_online(*resource*, *filter_type*)

to_dict()

See base class.

tear_down()

Releases system resources held by the connection.

class spinetoolbox.project_item.logging_connection.**LoggingJump**(**args*, *toolbox=None*, ***kwargs*)

Bases: *spinetoolbox.log_mixin.LogMixin*, spine_engine.project_item.connection.Jump

Represents a conditional jump between two project items.

Parameters

- **source_name** (*str*) – source project item's name
- **source_position** (*str*) – source anchor's position
- **destination_name** (*str*) – destination project item's name
- **destination_position** (*str*) – destination anchor's position
- **condition** (*dict*) – jump condition

property graphics_item

static item_type()

spinetoolbox.project_item.project_item

Contains base classes for project items and item factories.

Module Contents

Classes

ProjectItem	Class for project items that are not category nor root.

class spinetoolbox.project_item.project_item.**ProjectItem**(*name*, *description*, *x*, *y*, *project*)

Bases: *spinetoolbox.log_mixin.LogMixin*, *spinetoolbox.metaobject.MetaObject*

Class for project items that are not category nor root. These items can be executed, refreshed, and so on.

x

horizontal position in the screen

Type
float

y

vertical position in the screen

Type
float

Parameters

- **name** (*str*) – item name
- **description** (*str*) – item description
- **x** (*float*) – horizontal position on the scene
- **y** (*float*) – vertical position on the scene
- **project** (SpineToolboxProject) – project item's project

property project

property logger

abstract property executable_class

create_data_dir()

data_files()

Returns a list of files that are in the data directory.

abstract static item_type()

Item's type identifier string.

Returns
type string

Return type
str

make_signal_handler_dict()

Returns a dictionary of all shared signals and their handlers. This is to enable simpler connecting and disconnecting. Must be implemented in subclasses.

activate()

Restore selections and connect signals.

deactivate()

Save selections and disconnect signals.

restore_selections()

Restore selections into shared widgets when this project item is selected.

save_selections()

Save selections in shared widgets for this project item into instance variables.

_connect_signals()

Connect signals to handlers.

_disconnect_signals()

Disconnect signals from handlers and check for errors.

set_properties_ui(*properties_ui*)

Sets the properties tab widget for the item.

Note that this method expects the widget that is generated from the .ui files and initialized with the setupUi() method rather than the entire properties tab widget.

Parameters
properties_ui (*QWidget*) – item's properties UI

specification()

Returns the specification for this item.

undo_specification()

set_specification(*specification*)

Pushes a new SetItemSpecificationCommand to the toolbox' undo stack.

do_set_specification(*specification*)

Sets specification for this item. Removes specification if None given as argument.

Parameters
specification (*ProjectItemSpecification*) – specification of this item. None removes the specification.

set_icon(*icon*)

Sets the icon for the item.

Parameters
icon (ProjectItemIcon) – item's icon

get_icon()

Returns the graphics item representing this item in the scene.

_check_notifications()

Checks if exclamation icon notifications need to be set or cleared.

clear_notifications()

Clear all notifications from the exclamation icon.

add_notification(*text*)

Add a notification to the exclamation icon.

remove_notification(*text*)

Remove the first notification that includes given subtext.

clear_other_notifications(*text*)

Remove notifications that don't include the given subtext.

set_rank(*rank*)

Set rank of this item for displaying in the design view.

handle_execution_successful(*execution_direction*, *engine_state*)

Performs item dependent actions after the execution item has finished successfully.

Parameters

- **execution_direction** (*ExecutionDirection*) – ExecutionDirection.FORWARD or ExecutionDirection.BACKWARD
- **engine_state** – engine state after item's execution

resources_for_direct_successors()

Returns resources for direct successors.

These resources can include transient files that don't exist yet, or filename patterns. The default implementation returns an empty list.

Returns
a list of ProjectItemResources

Return type
list

resources_for_direct_predecessors()

Returns resources for direct predecessors.

These resources can include transient files that don't exist yet, or filename patterns. The default implementation returns an empty list.

Returns
a list of ProjectItemResources

Return type
list

_resources_to_predecessors_changed()

Notifies direct predecessors that item's resources have changed.

_resources_to_predecessors_replaced(*old*, *new*)

Notifies direct predecessors that item's resources have been replaced.

Parameters

- **old** (*list of ProjectItemResource*) – old resources
- **new** (*list of ProjectItemResource*) – new resources

upstream_resources_updated(*resources*)

Notifies item that resources from direct predecessors have changed.

Parameters
resources (*list of ProjectItemResource*) – new resources from upstream

replace_resources_from_upstream(*old*, *new*)

Replaces existing resources from direct predecessor by a new ones.

Parameters

- **old** (*list of ProjectItemResource*) – old resources
- **new** (*list of ProjectItemResource*) – new resources

_resources_to_successors_changed()

Notifies direct successors that item's resources have changed.

_resources_to_successors_replaced(*old*, *new*)

Notifies direct successors that one of item's resources has been replaced.

Parameters

- **old** (*list of ProjectItemResource*) – old resources
- **new** (*list of ProjectItemResource*) – new resources

downstream_resources_updated(*resources*)

Notifies item that resources from direct successors have changed.

Parameters
resources (*list of ProjectItemResource*) – new resources from downstream

replace_resources_from_downstream(*old*, *new*)

Replaces existing resources from direct successor by a new ones.

Parameters

- **old** (*list of ProjectItemResource*) – old resources
- **new** (*list of ProjectItemResource*) – new resources

item_dict()

Returns a dictionary corresponding to this item.

Returns
serialized project item

Return type
dict

static item_dict_local_entries()

Returns entries or 'paths' in item dict that should be stored in project's local data directory.

Returns
local data item dict entries

Return type
list of tuple of str

static parse_item_dict(*item_dict*)

Reads the information needed to construct the base ProjectItem class from an item dict.

Parameters
item_dict (*dict*) – an item dict

Returns
item's name, description as well as x and y coordinates

Return type
tuple

`copy_local_data`(*item_dict*)

Copies local data linked to a duplicated project item.

Parameters
item_dict (*dict*) – serialized item

`abstract static from_dict`(*name*, *item_dict*, *toolbox*, *project*)

Deserialized an item from item dict.

Parameters

- **name** (*str*) – item's name
- **item_dict** (*dict*) – serialized item
- **toolbox** (ToolboxUI) – the main window
- **project** (SpineToolboxProject) – a project

Returns
deserialized item

Return type
ProjectItem

`actions`()

Item specific actions.

Returns
item's actions

Return type
list of QAction

`rename`(*new_name*, *rename_data_dir_message*)

Renames this item.

If the project item needs any additional steps in renaming, override this method in subclass. See e.g. rename() method in DataStore class.

Parameters

- **new_name** (*str*) – New name
- **rename_data_dir_message** (*str*) – Message to show when renaming item's data directory

Returns
True if item was renamed successfully, False otherwise

Return type
bool

`open_directory`(*checked=False*)

Open this item's data directory in file explorer.

`tear_down`()

Tears down this item. Called both before closing the app and when removing the item from the project. Implement in subclasses to eg close all QMainWindows opened by this item.

set_up()

Sets up this item. Called when adding the item to the project. Implement in subclasses to eg recreate attributes destroyed by tear_down.

update_name_label()

Updates the name label on the properties widget, used when selecting an item and renaming the selected one.

notify_destination(*source_item*)

Informs an item that it has become the destination of a connection between two items.

The default implementation logs a warning message. Subclasses should reimplement this if they need more specific behavior.

Parameters
source_item (ProjectItem) – connection source item

static upgrade_v1_to_v2(*item_name*, *item_dict*)

Upgrades item's dictionary from v1 to v2.

Subclasses should reimplement this method if there are changes between version 1 and version 2.

Parameters

- **item_name** (*str*) – item's name
- **item_dict** (*dict*) – Version 1 item dictionary

Returns
Version 2 item dictionary

Return type
dict

static upgrade_v2_to_v3(*item_name*, *item_dict*, *project_upgrader*)

Upgrades item's dictionary from v2 to v3.

Subclasses should reimplement this method if there are changes between version 2 and version 3.

Parameters

- **item_name** (*str*) – item's name
- **item_dict** (*dict*) – Version 2 item dictionary
- **project_upgrader** (ProjectUpgrader) – Project upgrader class instance

Returns
Version 3 item dictionary

Return type
dict

spinetoolbox.project_item.project_item_factory

Contains base classes for project items and item factories.

Module Contents

Classes

ProjectItemFactory	Class for project item factories.

class spinetoolbox.project_item.project_item_factory.**ProjectItemFactory**

Class for project item factories.

abstract static item_class()

Returns the project item's class.

Returns
item's class

Return type
type

static is_deprecated()

Queries if item is deprecated.

Returns
True if item is deprecated, False otherwise

Return type
bool

abstract static icon()

Returns the icon resource path.

Returns
str

abstract static icon_color()

Returns the icon color.

Returns
icon's color

Return type
QColor

abstract static make_add_item_widget(*toolbox*, *x*, *y*, *specification*)

Returns an appropriate Add project item widget.

Parameters

- **toolbox** (ToolboxUI) – the main window
- **x** (*int*) – Icon coordinates
- **y** (*int*) – Icon coordinates
- **specification** (*ProjectItemSpecification*) – item's specification

Returns
QWidget

abstract static make_icon(*toolbox*)

Returns a ProjectItemIcon to use with given toolbox, for given project item.

Parameters
toolbox (ToolboxUI) –

Returns
item's icon

Return type
ProjectItemIcon

abstract static make_item(*name*, *item_dict*, *toolbox*, *project*)

Returns a project item constructed from the given `item_dict`.

Parameters

- **name** (*str*) – item's name
- **item_dict** (*dict*) – serialized project item
- **toolbox** (ToolboxUI) – Toolbox main window
- **project** (SpineToolboxProject) – the project the item belongs to

Returns
ProjectItem

abstract static make_properties_widget(*toolbox*)

Creates the item's properties tab widget.

Returns
item's properties tab widget

Return type
QWidget

abstract static make_specification_menu(*parent*, *index*)

Creates item specification's context menu.

Subclasses that do not support specifications can still raise `NotImplementedError`.

Parameters

- **parent** (*QWidget*) – menu's parent widget
- **index** (*QModelIndex*) – an index from specification model

Returns
specification's context menu

Return type
ItemSpecificationMenu

abstract static make_specification_editor(*toolbox*, *specification=None*, *item=None*, ***kwargs*)

Creates the item's specification widget.

Subclasses that do not support specifications can still raise `NotImplementedError`.

Parameters

- **toolbox** (ToolboxUI) – Toolbox main window

- **specification** (*ProjectItemSpecification, optional*) – a specification to show in the widget or None for a fresh start
- **item** (ProjectItem, *optional*) – a project item. If the specification is accepted, it is also set for this item
- ****kwargs** – parameters passed to the specification widget

Returns
item's specification widget

Return type
QWidget

static repair_specification(*toolbox*, *specification*)

Called right after a spec is added to the project. Finds if there's something wrong with the spec and proposes actions to fix it with help from toolbox.

Parameters

- **toolbox** (ToolboxUI) – Toolbox main window
- **specification** (*ProjectItemSpecification*) – a specification to check

spinetoolbox.project_item.specification_editor_window

Contains SpecificationEditorWindowBase and ChangeSpecPropertyCommand

Module Contents

Classes

UniqueCommandId	

CommandId	Enum where members are also (and must be) ints
ChangeSpecPropertyCommand	Command to set specification properties.
SpecificationEditorWindowBase	Base class for spec editors.
_SpecNameDescriptionToolbar	QToolBar for line edits and a hamburger menu.

Functions

prompt_to_save_changes(parent, settings, save_callback)	Prompts to save changes.

class spinetoolbox.project_item.specification_editor_window.**UniqueCommandId**

_NEXT_ID = 1

classmethod unique_id()

Returns a new unique command id.

Returns
unique id

Return type
int

class spinetoolbox.project_item.specification_editor_window.**CommandId**

Bases: enum.IntEnum

Enum where members are also (and must be) ints

Initialize self. See help(type(self)) for accurate signature.

NONE

NAME_UPDATE

DESCRIPTION_UPDATE

class spinetoolbox.project_item.specification_editor_window.**ChangeSpecPropertyCommand**(*callback*, *new_value*, *old_value*, *cmd_name*, *command_id=CommandId*)

Bases: PySide6.QtGui.QUndoCommand

Command to set specification properties.

Parameters

- **callback** (*Callable*) – Function to call to set the spec property.
- **new_value** (*Any*) – new value
- **old_value** (*Any*) – old value
- **cmd_name** (*str*) – command name
- **command_id** (*IntEnum*) – command id

redo()

undo()

id()

mergeWith(*other*)

class spinetoolbox.project_item.specification_editor_window.**SpecificationEditorWindowBase**(*toolbox*, *specification=None*, *item=None*)

Bases: PySide6.QtWidgets.QMainWindow

Base class for spec editors.

Parameters

- **toolbox** (ToolboxUI) – QMainWindow instance
- **specification** (*ProjectItemSpecification, optional*) – If given, the form is prefilled with this specification

- **item** (ProjectItem, *optional*) – Sets the spec for this item if accepted

abstract property settings_group

Returns the settings group for this spec type.

Returns
str

property _duplicate_kwargs

abstract _make_ui()

Returns the ui object from Qt designer.

Returns
object

_restore_dock_widgets()

Restores dockWidgets to some default state. Called in the constructor, before restoring the ui from settings. Reimplement in subclasses if needed.

abstract _make_new_specification(*spec_name*)

Returns a ProjectItemSpecification from current form settings.

Parameters
spec_name (*str*) – Name of the spec

Returns
ProjectItemSpecification

spec_toolbar()

Returns spec editor window's toolbar, which contains e.g. the hamburger menu.

show_error(*message*)

_show_status_bar_msg(*msg*)

_populate_main_menu()

_update_window_modified(*clean*)

_set_window_title(*title*)

Sets window title.

Parameters
title (*str*) – new window title

_save(*exiting=None*)

Saves spec.

Parameters
exiting (*bool, optional*) – Set as True if called when trying to exit the editor window

Returns
True if operation was successful, False otherwise

Return type
bool

prompt_exit_without_saving()

Prompts whether the user wants to exit without saving or cancel the exit.

Returns
False if the user chooses to cancel, in which case we don't close the form.

Return type
bool

_duplicate()

tear_down()

closeEvent(*event*)

class spinetoolbox.project_item.specification_editor_window.**_SpecNameDescriptionToolbar**(*parent*, *spec*, *undo_stack*)

Bases: PySide6.QtWidgets.QToolBar

QToolBar for line edits and a hamburger menu.

Parameters

- **parent** (*QMainWindow*) – QMainWindow instance
- **spec** (*ProjectItemSpecification*) – specification that is being edited
- **undo_stack** (*QUndoStack*) – an undo stack

name_changed

_make_main_menu()

_update_name(*name*)

Pushes a command to undo stack that updates the specification name.

Parameters
name (*str*) – updated name

_finish_name_editing()

Seals the last undo command.

_update_description(*description*)

Pushes a command to undo stack that updates the specification description.

Parameters
description (*str*) – updated description

_finish_description_editing()

Seals the last undo command.

do_set_name(*name*)

do_set_description(*description*)

name()

description()

spinetoolbox.project_item.specification_editor_window.**prompt_to_save_changes**(*parent*, *settings*, *save_callback*, *exiting=None*)

Prompts to save changes.

Parameters

- **parent** (*QWidget*) – Spec editor widget
- **settings** (*QSettings*) – Toolbox settings
- **save_callback** (*Callable*) – A function to call if the user chooses Save. It must return True or False depending on the outcome of the 'saving'.
- **exiting** (*bool, optional*) – Set as True if called when trying to exit the editor window

Returns
False if the user chooses to cancel, in which case we don't close the form.

Return type
bool

spinetoolbox.server

Package for handling the client part of executing projects on Spine Engine Server.

Submodules

spinetoolbox.server.engine_client

Client for exchanging messages between the toolbox and the Spine Engine Server.

Module Contents

Classes

ClientSecurityModel	Generic enumeration.
EngineClient	**param host** IP address of the Spine Engine Server

class spinetoolbox.server.engine_client.**ClientSecurityModel**

Bases: enum.Enum

Generic enumeration.

Derive from this class to define new enumerations.

NONE = 0

STONEHOUSE = 1

class spinetoolbox.server.engine_client.**EngineClient**(*host*, *port*, *sec_model*, *sec_folder*, *ping=True*)

Parameters

- **host** (*str*) – IP address of the Spine Engine Server
- **port** (*int*) – Port of the client facing (frontend) socket on Spine Engine Server

- **sec_model** (ClientSecurityModel) – Client security scheme
- **sec_folder** (*str*) – Path to security file directory
- **ping** (*bool*) – Whether to check connectivity at instance creation

connect_pull_socket(*port*)

Connects a PULL socket for receiving engine execution events and files from server.

Parameters
port (*str*) – Port of the PUSH socket on server

rcv_next(*dealer_or_pull*)

Polls all sockets and returns a new reply based on given socket 'name'.

Parameters
dealer_or_pull (*str*) – "dealer" to wait reply from DEALER socket, "pull" to wait reply from PULL socket

_check_connectivity(*timeout*)

Pings server, waits for the response, and acts accordingly.

Parameters
timeout (*int*) – Time to wait for a response before giving up [ms]

Returns
void

Raises
RemoteEngineInitFailed if the server is not responding. –

set_start_time()

Sets a start time for an operation. Call get_elapsed_time() after an operation has finished to get the elapsed time string.

upload_project(*project_dir_name*, *fpath*)

Uploads the zipped project file to server. Project zip file must be ready and the server available before calling this method.

Parameters

- **project_dir_name** (*str*) – Project directory name
- **fpath** (*str*) – Absolute path to zipped project file.

Returns
Project execution job Id

Return type
str

start_execution(*engine_data*, *job_id*)

Sends the start execution request along with job Id and engine (dag) data to the server. Response message data contains the push/pull socket port if execution starts successfully.

Parameters

- **engine_data** (*str*) – Input for SpineEngine as JSON str. Includes most of project.json, settings, etc.
- **job_id** (*str*) – Project execution job Id on server

Returns
Response tuple (event_type, data). Event_type is "server_init_failed", "remote_execution_init_failed" or "remote_execution_started. data is an error message or the publish and push sockets ports concatenated with ':'.

Return type
tuple

`stop_execution`(*job_id*)

Sends a request to stop executing the DAG that is managed by this client.

Parameters
job_id (*str*) – Job Id on server to stop

`answer_prompt`(*job_id*, *prompter_id*, *answer*)

Sends a request to answer a prompt from the DAG that is managed by this client.

Parameters

- **job_id** (*str*) – Job Id on server to stop
- **prompter_id** (*int*) –
- **answer** –

`download_files`(*q*)

Pulls files from server until b'END' is received.

`save_downloaded_file`(*b_rel_path*, *file_data*)

Saves downloaded file to project directory.

Parameters

- **b_rel_path** (*bytes*) – Relative path (to project dir) where the file should be saved
- **file_data** (*bytes*) – File as bytes object

`retrieve_project`(*job_id*)

Retrieves a zipped project file from server.

Parameters
job_id (*str*) – Job Id for finding the project directory on server

Returns
Zipped project file

Return type
bytes

`remove_project_from_server`(*job_id*)

Sends a request to remove a project directory from server.

Parameters
job_id (*str*) – Job Id for finding the project directory on server

Returns
Message from server

Return type
str

`send_is_complete`(*persistent_key*, *cmd*)

Sends a request to process is_complete(cmd) in persistent manager on server and returns the response.

send_issue_persistent_command(*persistent_key*, *cmd*)

Sends a request to process given command in persistent manager identified by given key. Yields the response string(s) as they arrive from server.

send_get_persistent_completions(*persistent_key*, *text*)

Requests completions to given text from persistent execution backend.

send_get_persistent_history_item(*persistent_key*, *text*, *prefix*, *backwards*)

Requests the former or latter history item from persistent execution backend.

send_restart_persistent(*persistent_key*)

Sends restart persistent cmd to persistent execution manager backend on server. Yields the messages resulting from this operation to persistent console client.

send_interrupt_persistent(*persistent_key*)

Sends interrupt persistent cmd to persistent execution manager backend on server.

send_kill_persistent(*persistent_key*)

Sends kill persistent cmd to persistent execution manager backend on server.

Parameters
persistent_key (*tuple*) – persistent manager identifier

send_request_to_persistent(*data*)

Sends given data containing persistent_key, command, cmd_to_persistent to Spine Engine Server to be processed by a persistent execution manager backend. Makes a request using REQ socket, parses the response into a ServerMessage, and returns the second part of the data field.

send_request_to_persistent_generator(*data*)

Pulls all messages from server, that were the result of sending given data to Spine Engine Server.

get_elapsed_time()

Returns the elapsed time between now and when self.start_time was set.

Returns
Time string with unit(s)

Return type
str

close()

Closes client sockets, context and thread.

spinetoolbox.spine_db_editor

This subpackage contains GUI files for the Spine db editor.

Subpackages

spinetoolbox.spine_db_editor.mvcmodels

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

Submodules

spinetoolbox.spine_db_editor.mvcmodels.alternative_item

Classes to represent items in an alternative tree.

Module Contents

Classes

DBItem	A root item representing a db.
AlternativeItem	An alternative leaf item.

Attributes

_ALTERNATIVE_ICON

spinetoolbox.spine_db_editor.mvcmodels.alternative_item.**_ALTERNATIVE_ICON** = '\uf277'

class spinetoolbox.spine_db_editor.mvcmodels.alternative_item.**DBItem**(**args*, ***kwargs*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.EmptyChildMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.FetchMoreMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.StandardDBItem*

A root item representing a db.

property item_type

property fetch_item_type

empty_child()

_make_child(*id_*)

class spinetoolbox.spine_db_editor.mvcmodels.alternative_item.**AlternativeItem**(*model*, *identifier=None*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.GrayIfLastMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.EditableMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.LeafItem*

An alternative leaf item.

Parameters

- **model** (MinimalTreeModel) –
- **identifier** (*int, optional*) – item's database id

property item_type

property icon_code

tool_tip(*column*)

add_item_to_db(*db_item*)

update_item_in_db(*db_item*)

flags(*column*)

Makes items editable.

spinetoolbox.spine_db_editor.mvcmodels.alternative_model

Contains alternative tree model.

Module Contents

Classes

AlternativeModel	A model to display alternatives in a tree view.

class spinetoolbox.spine_db_editor.mvcmodels.alternative_model.**AlternativeModel**(*db_editor*, *db_mngr*, **db_maps*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_model_base.TreeModelBase*

A model to display alternatives in a tree view.

Parameters

- **db_editor** (SpineDBEditor) –
- **db_mngr** (SpineDBManager) –
- ***db_maps** – DatabaseMapping instances

_make_db_item(*db_map*)

mimeData(*indexes*)

Stores selected indexes into MIME data.

The MIME data structure contains two distinct data:

- Text representation of the selection
- A pickled dict mapping db identifier to list of alternative ids

Parameters

indexes (*Sequence of QModelIndex*) – selected indexes

Returns
MIME data

Return type
QMimeData

paste_alternative_mime_data(*mime_data*, *database_item*)

Pastes alternatives from mime data into model.

Parameters

- **mime_data** (*QMimeData*) – mime data
- **database_item** (alternative_item.DBItem) – target database item

spinetoolbox.spine_db_editor.mvcmodels.colors

Color constants for models.

Module Contents

spinetoolbox.spine_db_editor.mvcmodels.colors.**PIVOT_TABLE_HEADER_COLOR**

spinetoolbox.spine_db_editor.mvcmodels.colors.**FIXED_FIELD_COLOR**

spinetoolbox.spine_db_editor.mvcmodels.colors.**SELECTED_COLOR**

spinetoolbox.spine_db_editor.mvcmodels.compound_models

Compound models. These models concatenate several 'single' models and one 'empty' model.

Module Contents

Classes

CompoundModelBase	A base model for all models that show data in stacked format.
FilterEntityAlternativeMixin	Provides the interface to filter by entity and alternative.
EditParameterValueMixin	Provides the interface to edit values via ParameterValueEditor.
CompoundParameterDefinitionModel	A model that concatenates several single parameter_definition models and one empty parameter_definition model.
CompoundParameterValueModel	A model that concatenates several single parameter_value models and one empty parameter_value model.
CompoundEntityAlternativeModel	Provides the interface to filter by entity and alternative.

class spinetoolbox.spine_db_editor.mvcmodels.compound_models.**CompoundModelBase**(*parent*, *db_mngr*, **db_maps*)

Bases: *spinetoolbox.mvcmodels.compound_table_model.CompoundWithEmptyTableModel*

A base model for all models that show data in stacked format.

Parameters

- **parent** (SpineDBEditor) – the parent object
- **db_mngr** (SpineDBManager) – the database manager
- ***db_maps** (*DatabaseMapping*) – the database maps included in the model

property field_map

abstract property item_type

Returns the DB item type, e.g., 'parameter_value'.

Returns
str

abstract property _single_model_type

Returns a constructor for the single models.

Returns
SingleParameterModel

abstract property _empty_model_type

Returns a constructor for the empty model.

Returns
EmptyParameterModel

abstract _make_header()

canFetchMore(*_parent*)

Returns True if any of the submodels that haven't been fetched yet can fetch more.

fetchMore(*_parent*)

Fetches the next sub model and increments the fetched counter.

shows_item(*item*, *db_map*)

reset_db_maps(*db_maps*)

init_model()

Initializes the model.

get_auto_filter_menu(*logical_index*)

Returns auto filter menu for given logical index from header view.

Parameters
logical_index (*int*) –

Returns
AutoFilterMenu

_make_auto_filter_menu(*field*)

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

Returns an italic font in case the given column has an autofilter installed.

_create_empty_model()

Returns the empty model for this compound model.

Returns
EmptyParameterModel

filter_accepts_model(*model*)

Returns a boolean indicating whether the given model passes the filter for compound model.

Parameters
model (SingleModelBase *or* EmptyModelBase) –

Returns
bool

_class_filter_accepts_model(*model*)

_auto_filter_accepts_model(*model*)

accepted_single_models()

Returns a list of accepted single models by calling filter_accepts_model on each of them, just for convenience.

Returns
list

_invalidate_filter()

Sets the filter invalid.

stop_invalidating_filter()

Stops invalidating the filter.

set_filter_class_ids(*class_ids*)

clear_auto_filter()

set_auto_filter(*field*, *values*)

Updates and applies the auto filter.

Parameters

- **field** (*str*) – the field name
- **values** (*dict*) – mapping (db_map, entity_class_id) to set of valid values

_set_compound_auto_filter(*field*, *values*)

Sets the auto filter for given column in the compound model.

Parameters

- **field** (*str*) – the field name
- **values** (*set*) – set of valid (db_map, item_type, id) tuples

_set_single_auto_filter(*model*, *field*)

Sets the auto filter for given column in the given single model.

Parameters

- **model** (*SingleParameterModel*) – the model

- **field** (*str*) – the field name

Returns
True if the auto-filtered values were updated, None otherwise

Return type
bool

_row_map_iterator_for_model(*model*)

Yields row map for the given model. Reimplemented to take filter status into account.

Parameters
model (*SingleParameterModel, EmptyParameterModel*) –

Yields
tuple – (model, row number) for each accepted row

_models_with_db_map(*db_map*)

Returns a collection of single models with given db_map.

Parameters
db_map (*DatabaseMapping*) –

Returns
list

static _items_per_class(*items*)

Returns a dict mapping entity_class ids to a set of items.

Parameters
items (*list*) –

Returns
dict

handle_items_added(*db_map_data*)

Runs when either parameter definitions or values are added to the dbs. Adds necessary sub-models and initializes them with data. Also notifies the empty model so it can remove rows that are already in.

Parameters
db_map_data (*dict*) – list of added dict-items keyed by DatabaseMapping

_get_insert_position(*model*)

_create_single_model(*db_map*, *entity_class_id*, *committed*)

_add_items(*db_map*, *entity_class_id*, *ids*, *committed*)

Creates new single model and resets it with the given parameter ids.

Parameters

- **db_map** (*DatabaseMapping*) – database map
- **entity_class_id** (*int*) – parameter's entity class id
- **ids** (*list of int*) – parameter ids
- **committed** (*bool*) – True if the ids have been committed, False otherwise

handle_items_updated(*db_map_data*)

Runs when either parameter definitions or values are updated in the dbs. Emits dataChanged so the parameter_name column is refreshed.

Parameters
db_map_data (*dict*) – list of updated dict-items keyed by DatabaseMapping

handle_items_removed(*db_map_data*)

Runs when either parameter definitions or values are removed from the dbs. Removes the affected rows from the corresponding single models.

Parameters
db_map_data (*dict*) – list of removed dict-items keyed by DatabaseMapping

db_item(*index*)

db_map_id(*index*)

get_entity_class_id(*index*, *db_map*)

filter_by(*rows_per_column*)

filter_excluding(*rows_per_column*)

class spinetoolbox.spine_db_editor.mvcmodels.compound_models.**FilterEntityAlternativeMixin**(**args*, ***kwargs*)

Provides the interface to filter by entity and alternative.

init_model()

set_filter_entity_ids(*entity_ids*)

set_filter_alternative_ids(*alternative_ids*)

_create_single_model(*db_map*, *entity_class_id*, *committed*)

class spinetoolbox.spine_db_editor.mvcmodels.compound_models.**EditParameterValueMixin**

Provides the interface to edit values via ParameterValueEditor.

index_name(*index*)

Generates a name for data at given index.

Parameters
index (*QModelIndex*) – index to model

Returns
label identifying the data

Return type
str

get_set_data_delayed(*index*)

Returns a function that ParameterValueEditor can call to set data for the given index at any later time, even if the model changes.

Parameters
index (*QModelIndex*) –

Returns
function

class spinetoolbox.spine_db_editor.mvcmodels.compound_models.**CompoundParameterDefinitionModel**(*parent*, *db_mngr*, **db_maps*)

Bases: *EditParameterValueMixin*, *CompoundModelBase*

A model that concatenates several single parameter_definition models and one empty parameter_definition model.

Parameters

- **parent** (SpineDBEditor) – the parent object
- **db_mngr** (SpineDBManager) – the database manager
- ***db_maps** (*DatabaseMapping*) – the database maps included in the model

property item_type

Returns the DB item type, e.g., 'parameter_value'.

Returns
str

property field_map

property _single_model_type

Returns a constructor for the single models.

Returns
SingleParameterModel

property _empty_model_type

Returns a constructor for the empty model.

Returns
EmptyParameterModel

_make_header()

class spinetoolbox.spine_db_editor.mvcmodels.compound_models.**CompoundParameterValueModel**(**args*, ***kwargs*)

Bases: *FilterEntityAlternativeMixin*, *EditParameterValueMixin*, *CompoundModelBase*

A model that concatenates several single parameter_value models and one empty parameter_value model.

property item_type

Returns the DB item type, e.g., 'parameter_value'.

Returns
str

property field_map

property _single_model_type

Returns a constructor for the single models.

Returns
SingleParameterModel

property _empty_model_type

Returns a constructor for the empty model.

Returns
EmptyParameterModel

_make_header()

class spinetoolbox.spine_db_editor.mvcmodels.compound_models.**CompoundEntityAlternativeModel**(**args*, ***kwargs*)

Bases: *FilterEntityAlternativeMixin*, *CompoundModelBase*

Provides the interface to filter by entity and alternative.

property item_type

Returns the DB item type, e.g., 'parameter_value'.

Returns
str

property _single_model_type

Returns a constructor for the single models.

Returns
SingleParameterModel

property _empty_model_type

Returns a constructor for the empty model.

Returns
EmptyParameterModel

_make_header()

spinetoolbox.spine_db_editor.mvcmodels.empty_models

Empty models for parameter definitions and values.

Module Contents

Classes

EmptyModelBase	Base class for all empty models that go in a CompoundModelBase subclass.
ParameterMixin	
EntityMixin	
EmptyParameterDefinitionModel	An empty parameter_definition model.
EmptyParameterValueModel	An empty parameter_value model.
EmptyEntityAlternativeModel	Makes relationships on the fly.

class spinetoolbox.spine_db_editor.mvcmodels.empty_models.**EmptyModelBase**(*parent*)

Bases: *spinetoolbox.mvcmodels.empty_row_model.EmptyRowModel*

Base class for all empty models that go in a CompoundModelBase subclass.

Parameters
parent (CompoundModelBase) – the parent model

abstract property item_type

property field_map

property can_be_filtered

add_items_to_db(*db_map_data*)

Add items to db.

Parameters
db_map_data (*dict*) – mapping DiffDatabaseMapping instance to list of items

abstract _make_unique_id(*item*)

Returns a unique id for the given model item (name-based). Used by handle_items_added to identify which rows have been added and thus need to be removed.

accepted_rows()

db_item(*_index*)

item_id(*_row*)

handle_items_added(*db_map_data*)

Runs when parameter definitions or values are added. Finds and removes model items that were successfully added to the db.

batch_set_data(*indexes*, *data*)

Sets data for indexes in batch. If successful, add items to db.

_autocomplete_row(*db_map*, *item*)

Fills in entity_class_name whenever other selections make it obvious.

abstract _entity_class_name_candidates(*db_map*, *item*)

_make_item(*row*)

_make_db_map_data(*rows*)

Returns model data grouped by database map.

Parameters
rows (*set*) – group data from these rows

Returns
mapping DiffDatabaseMapping instance to list of items

Return type
dict

data(*index*, *role=Qt.ItemDataRole.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.

class spinetoolbox.spine_db_editor.mvcmodels.empty_models.**ParameterMixin**

property value_field

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

static _entity_class_name_candidates_by_parameter(*db_map*, *item*)

class spinetoolbox.spine_db_editor.mvcmodels.empty_models.**EntityMixin**

abstract _do_add_items_to_db(*db_map_items*)

add_items_to_db(*db_map_data*)

Overriden to add entities on the fly first.

_make_item(*row*)

static _entity_class_name_candidates_by_entity(*db_map*, *item*)

class spinetoolbox.spine_db_editor.mvcmodels.empty_models.**EmptyParameterDefinitionModel**(*parent*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.single_and_empty_model_mixins.SplitValueAndTypeMixin*, *ParameterMixin*, *EmptyModelBase*

An empty parameter_definition model.

Parameters
parent (CompoundModelBase) – the parent model

property item_type

_make_unique_id(*item*)

Returns a unique id for the given model item (name-based). Used by handle_items_added to identify which rows have been added and thus need to be removed.

static _check_item(*item*)

Checks if a db item is ready to be inserted.

_entity_class_name_candidates(*db_map*, *item*)

_do_add_items_to_db(*db_map_items*)

class spinetoolbox.spine_db_editor.mvcmodels.empty_models.**EmptyParameterValueModel**(*parent*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.single_and_empty_model_mixins.MakeEntityOnTheFlyMixin*, *spinetoolbox.spine_db_editor.mvcmodels.single_and_empty_model_mixins.SplitValueAndTypeMixin*, *ParameterMixin*, *EntityMixin*, *EmptyModelBase*

An empty parameter_value model.

Parameters
parent (CompoundModelBase) – the parent model

property item_type

static _check_item(*item*)

Checks if a db item is ready to be inserted.

_make_unique_id(*item*)

Returns a unique id for the given model item (name-based). Used by handle_items_added to identify which rows have been added and thus need to be removed.

_do_add_items_to_db(*db_map_items*)

_entity_class_name_candidates(*db_map*, *item*)

class spinetoolbox.spine_db_editor.mvcmodels.empty_models.**EmptyEntityAlternativeModel**(*parent*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.single_and_empty_model_mixins.MakeEntityOnTheFlyMixin*, *EntityMixin*, *EmptyModelBase*

Makes relationships on the fly.

Parameters
parent (CompoundModelBase) – the parent model

property item_type

static _check_item(*item*)

Checks if a db item is ready to be inserted.

_make_unique_id(*item*)

Returns a unique id for the given model item (name-based). Used by handle_items_added to identify which rows have been added and thus need to be removed.

_do_add_items_to_db(*db_map_items*)

_entity_class_name_candidates(*db_map*, *item*)

spinetoolbox.spine_db_editor.mvcmodels.entity_tree_item

Classes to represent entities in a tree.

Module Contents

Classes

EntityClassIndex	dict() -> new empty dictionary
EntityGroupIndex	dict() -> new empty dictionary
EntityIndex	dict() -> new empty dictionary
EntityTreeRootItem	A tree item that may belong in multiple databases.
EntityClassItem	An entity_class item.
EntityItem	An entity item.

class spinetoolbox.spine_db_editor.mvcmodels.entity_tree_item.**EntityClassIndex**

Bases: *spinetoolbox.fetch_parent.FetchIndex*

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's

(key, value) pairs

dict(iterable) -> new dictionary initialized as if via:
d = {} for k, v in iterable:

d[k] = v

dict(kwargs) -> new dictionary initialized with the name=value pairs**
in the keyword argument list. For example: dict(one=1, two=2)

Initialize self. See help(type(self)) for accurate signature.

process_item(*item*, *db_map*)

class spinetoolbox.spine_db_editor.mvcmodels.entity_tree_item.**EntityGroupIndex**

Bases: *spinetoolbox.fetch_parent.FetchIndex*

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's

(key, value) pairs

dict(iterable) -> new dictionary initialized as if via:
d = {} for k, v in iterable:

d[k] = v

dict(kwargs) -> new dictionary initialized with the name=value pairs**
in the keyword argument list. For example: dict(one=1, two=2)

Initialize self. See help(type(self)) for accurate signature.

process_item(*item*, *db_map*)

class spinetoolbox.spine_db_editor.mvcmodels.entity_tree_item.**EntityIndex**

Bases: *spinetoolbox.fetch_parent.FetchIndex*

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's

(key, value) pairs

dict(iterable) -> new dictionary initialized as if via:
d = {} for k, v in iterable:

d[k] = v

dict(kwargs) -> new dictionary initialized with the name=value pairs**
in the keyword argument list. For example: dict(one=1, two=2)

Initialize self. See help(type(self)) for accurate signature.

process_item(*item*, *db_map*)

class spinetoolbox.spine_db_editor.mvcmodels.entity_tree_item.**EntityTreeRootItem**(**args*, ***kwargs*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.multi_db_tree_item.MultiDBTreeItem*

A tree item that may belong in multiple databases.

Parameters

- **model** (*MinimalTreeModel, optional*) – item's model
- **db_map_ids** (*dict, optional*) – maps instances of DatabaseMapping to the id of the item in that db

property visible_children

property display_id

See super class.

property display_icon

Returns an icon to display next to the name. Reimplement in subclasses to return something nice.

property display_data

See super class.

property child_item_class

Returns ObjectClassItem.

item_type = 'root'

set_data(*column*, *value*, *role*)

See base class.

_polish_children(*children*)

See base class.

class spinetoolbox.spine_db_editor.mvcmodels.entity_tree_item.**EntityClassItem**(*model*, *db_map_ids=None*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.multi_db_tree_item.MultiDBTreeItem*

An entity_class item.

Parameters

- **model** (*MinimalTreeModel, optional*) – item's model
- **db_map_ids** (*dict, optional*) – maps instances of DatabaseMapping to the id of the item in that db

property display_icon

Returns class icon.

property child_item_class

Returns the type of child items.

property _children_sort_key

Reimplemented so groups are above non-groups.

property display_data

Returns the name for display.

property has_dimensions

visual_key = ['name', 'dimension_name_list', 'superclass_name']

item_type = 'entity_class'

_fetch_index

is_hidden()

default_parameter_data()

Return data to put as default in a parameter table when this item is selected.

data(*column*, *role=Qt.ItemDataRole.DisplayRole*)

Returns data for given column and role.

_key_for_index(*db_map*)

accepts_item(*item*, *db_map*)

set_data(*column*, *value*, *role*)

See base class.

_polish_children(*children*)

See base class.

class spinetoolbox.spine_db_editor.mvcmodels.entity_tree_item.**EntityItem**(**args*, *is_member=False*, ***kwargs*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.multi_db_tree_item.MultiDBTreeItem*

An entity item.

Parameters

- **model** (*MinimalTreeModel, optional*) – item's model
- **db_map_ids** (*dict, optional*) – maps instances of DatabaseMapping to the id of the item in that db

property is_group

property child_item_class

Child class is always *EntityItem*.

property display_icon

Returns corresponding class icon.

property element_name_list

property element_byname_list

property byname

property entity_class_name

property entity_class_key

property display_data

Returns the name for display.

property edit_data

visual_key = ['entity_class_name', 'entity_byname']

item_type = 'entity'

_fetch_index

_entity_group_index

data(*column*, *role=Qt.ItemDataRole.DisplayRole*)

Returns data for given column and role.

set_data(*column*, *value*, *role*)

See base class.

default_parameter_data()

Return data to put as default in a parameter table when this item is selected.

is_valid()

See base class.

Additionally, checks that the parent entity (if any) is still an element in this entity.

_can_fetch_more_entity_groups()

can_fetch_more()

Returns whether this item can fetch more.

_fetch_more_entity_groups()

fetch_more()

Fetches children from all associated databases.

_key_for_index(*db_map*)

_key_for_entity_group_index(*db_map*)

accepts_item(*item*, *db_map*)

_accepts_entity_group_item(*item*, *db_map*)

_handle_entity_group_items_added(*db_map_data*)

_handle_entity_group_items_removed(*db_map_data*)

tear_down()

Do stuff after the item has been removed.

spinetoolbox.spine_db_editor.mvcmodels.entity_tree_models

Models to represent entities in a tree.

Module Contents

Classes

EntityTreeModel	Base class for all tree models in Spine db editor.

Functions

group_items_by_db_map(indexes)	Groups items from given tree indexes by db map.

class spinetoolbox.spine_db_editor.mvcmodels.entity_tree_models.**EntityTreeModel**(**args*, ***kwargs*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.multi_db_tree_model.MultiDBTreeModel*

Base class for all tree models in Spine db editor.

Init class.

Parameters

- **db_editor** (SpineDBEditor) –
- **db_mngr** (SpineDBManager) – A manager for the given db_maps
- ***db_maps** – DatabaseMapping instances

property root_item_type

Implement in subclasses to create a model specific to any entity type.

property hide_empty_classes

find_next_entity_index(*index*)

Find and return next occurrence of relationship item.

save_hide_empty_classes()

spinetoolbox.spine_db_editor.mvcmodels.entity_tree_models.**group_items_by_db_map**(*indexes*)

Groups items from given tree indexes by db map.

Parameters
indexes (*Iterable of QModelIndex*) – index to entity tree model

Returns
lists of dictionary items keyed by DatabaseMapping

Return type
dict

spinetoolbox.spine_db_editor.mvcmodels.frozen_table_model

Contains FrozenTableModel class.

Module Contents

Classes

FrozenTableModel	Used by custom_qtableview.FrozenTableView

class spinetoolbox.spine_db_editor.mvcmodels.frozen_table_model.**FrozenTableModel**(*db_mngr*, *parent=None*)

Bases: PySide6.QtCore.QAbstractTableModel

Used by custom_qtableview.FrozenTableView

Parameters

- **db_mngr** (SpineDBManager) – database manager
- **parent** (*QObject, optional*) – parent object

property headers

selected_row_changed

set_headers(*headers*)

Sets headers for the header row wiping data.

This method does nothing if the new headers are equal to existing ones.

Parameters
headers (*Iterable of str*) – headers

Returns
True if model was reset, False otherwise

Return type
bool

clear_model()

add_values(*data*)

Adds more frozen values that aren't in the table already.

Parameters
data (*set of tuple*) – frozen values

remove_values(*data*)

Removes frozen values from the table.

Parameters
data (*set of tuple*) – frozen values

clear_selected()

Clears selected row.

set_selected(*row*)

Changes selected row.

Parameters
row (*int*) – row index

get_selected()

Returns selected row.

Returns
row index or None if no row is selected

Return type
int

get_frozen_value()

Return currently selected frozen value.

Returns
frozen value

Return type
tuple

rowCount(*parent=QModelIndex()*)

columnCount(*parent=QModelIndex()*)

row(*index*)

insert_column_data(*header*, *values*, *column*)

Inserts new column with given header.

Parameters

- **header** (*str*) – frozen header
- **values** (*set of tuple*) – column's values
- **column** (*int*) – position

remove_column(*column*)

Removes column and makes rows unique.

Parameters
column (*int*) – column to remove

moveColumns(*sourceParent*, *sourceColumn*, *count*, *destinationParent*, *destinationChild*)

_keep_sorted(*update_selected_row=True*)

Sorts the data table.

_unique_values()

Turns non-header data into sets of unique values on each column.

Returns
each column's unique values

Return type
list of set

_find_first(*row_data*, *mask_column=None*)

Finds first row that matches given row data.

Parameters

- **row_data** (*tuple*) – row data to search for
- **mask_column** (*int, optional*) – ignored column

Returns
row index

Return type
int

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

_tooltip_from_data(*row*, *column*)

Resolves item tooltip which is usually its description.

Parameters

- **row** (*int*) – row
- **column** (*int*) – column

Returns
value's tooltip

Return type
str

_name_from_data(*value*, *header*)

Resolves item name.

Parameters

- **value** (*tuple or DatabaseMapping*) – cell value
- **header** (*str*) – column header

Returns
value's name

Return type
str

spinetoolbox.spine_db_editor.mvcmodels.item_metadata_table_model

Contains *ItemMetadataTableModel* and associated functionality.

Module Contents

Classes

ExtraColumn	Identifiers for hidden table columns.
ItemType	Allowed item types.
ItemMetadataTableModel	Model for entity and parameter value metadata.

class spinetoolbox.spine_db_editor.mvcmodels.item_metadata_table_model.**ExtraColumn**

Bases: enum.IntEnum

Identifiers for hidden table columns.

Initialize self. See help(type(self)) for accurate signature.

ITEM_METADATA_ID

METADATA_ID

class spinetoolbox.spine_db_editor.mvcmodels.item_metadata_table_model.**ItemType**

Bases: enum.Enum

Allowed item types.

ENTITY

VALUE

class spinetoolbox.spine_db_editor.mvcmodels.item_metadata_table_model.**ItemMetadataTableModel**(*db_mngr*, *db_maps*, *db_editor*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.metadata_table_model_base.MetadataTableModelBase*

Model for entity and parameter value metadata.

Parameters

- **db_mngr** (SpineDBManager) – database manager
- **db_maps** (*Iterable of DatabaseMapping*) – database maps
- **db_editor** (SpineDBEditor) – DB editor

_ITEM_NAME_KEY = 'metadata_name'

_ITEM_VALUE_KEY = 'metadata_value'

_fetch_parents()

Yields fetch parents for this model.

Yields
FetchParent

clear()

Clears the model.

static _make_hidden_adder_columns()

See base class.

_accepts_entity_metadata_item(*item*, *db_map*)

_accepts_parameter_value_metadata_item(*item*, *db_map*)

set_entity_ids(*db_map_ids*)

Sets the model to show metadata from given entity.

Parameters
db_map_ids (*dict*) – mapping from database mapping to entity's id in that database

set_parameter_value_ids(*db_map_ids*)

Sets the model to show metadata from given parameter value.

Parameters
db_map_ids (*dict*) – mapping from database mapping to value's id in that database

_reset_metadata(*item_type*, *db_map_ids*)

Resets model.

Parameters

- **item_type** (ItemType) – current item type
- **db_map_ids** (*dict*) – mapping from database mapping to value's id in that database

_reset_fetch_parents()

_add_data_to_db_mngr(*name*, *value*, *db_map*)

See base class.

_update_data_in_db_mngr(*id_*, *name*, *value*, *db_map*)

See base class

flags(*index*)

static _ids_from_added_item(*item*)

See base class.

static _extra_cells_from_added_item(*item*)

See base class.

_set_extra_columns(*row*, *ids*)

See base class.

_database_table_name()

See base class

_row_id(*row*)

See base class.

add_item_metadata(*db_map_data*)

Adds new item metadata from database manager to the model.

Parameters
db_map_data (*dict*) – added items keyed by database mapping

update_item_metadata(*db_map_data*)

Updates item metadata in model after it has been updated in databases.

Parameters
db_map_data (*dict*) – updated metadata records

remove_item_metadata(*db_map_data*)

Removes item metadata from model after it has been removed from databases.

Parameters
db_map_data (*dict*) – removed items keyed by database mapping

spinetoolbox.spine_db_editor.mvcmodels.metadata_table_model

Contains *MetadataTableModel* and associated functionality.

Module Contents

Classes

ExtraColumn	Identifiers for hidden table columns.
MetadataTableModel	Model for metadata.

class spinetoolbox.spine_db_editor.mvcmodels.metadata_table_model.**ExtraColumn**

Bases: enum.IntEnum

Identifiers for hidden table columns.

Initialize self. See help(type(self)) for accurate signature.

ID

class spinetoolbox.spine_db_editor.mvcmodels.metadata_table_model.**MetadataTableModel**(*db_mngr*, *db_maps*, *db_editor*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.metadata_table_model_base.MetadataTableModelBase*

Model for metadata.

Parameters

- **db_mngr** (SpineDBManager) – database manager
- **db_maps** (*Iterable of DatabaseMapping*) – database maps
- **db_editor** (SpineDBEditor) – DB editor

_ITEM_NAME_KEY = 'name'

_ITEM_VALUE_KEY = 'value'

static _make_hidden_adder_columns()

See base class.

_add_data_to_db_mngr(*name*, *value*, *db_map*)

See base class.

_update_data_in_db_mngr(*id_*, *name*, *value*, *db_map*)

See base class

_database_table_name()

See base class

_row_id(*row*)

See base class.

flags(*index*)

_fetch_parents()

Yields fetch parents for this model.

Yields
FetchParent

static _ids_from_added_item(*item*)

See base class.

static _extra_cells_from_added_item(*item*)

See base class.

_set_extra_columns(*row*, *ids*)

See base class.

add_metadata(*db_map_data*)

Adds new metadata from database manager to the model.

Parameters
db_map_data (*dict*) – added metadata items keyed by database mapping

update_metadata(*db_map_data*)

Updates model according to data received from database manager.

Parameters
db_map_data (*dict*) – updated metadata items keyed by database mapping

remove_metadata(*db_map_data*)

Removes metadata from model after it has been removed from databases.

Parameters
db_map_data (*dict*) – removed items keyed by database mapping

spinetoolbox.spine_db_editor.mvcmodels.metadata_table_model_base

Contains base class for metadata table models associated functionality.

Module Contents

Classes

Column	Identifiers for visible table columns.
MetadataTableModelBase	Base for metadata table models

Attributes

FLAGS_FIXED
FLAGS_EDITABLE

class spinetoolbox.spine_db_editor.mvcmodels.metadata_table_model_base.**Column**

Bases: enum.IntEnum

Identifiers for visible table columns.

Initialize self. See help(type(self)) for accurate signature.

NAME = 0

VALUE = 1

DB_MAP = 2

static max()

spinetoolbox.spine_db_editor.mvcmodels.metadata_table_model_base.**FLAGS_FIXED**

spinetoolbox.spine_db_editor.mvcmodels.metadata_table_model_base.**FLAGS_EDITABLE**

class spinetoolbox.spine_db_editor.mvcmodels.metadata_table_model_base.**MetadataTableModelBase**(*db_mngr*, *db_maps*, *db_editor*)

Bases: PySide6.QtCore.QAbstractTableModel

Base for metadata table models

Parameters

- **db_mngr** (SpineDBManager) – database manager
- **db_maps** (*Iterable of DatabaseMapping*) – database maps
- **db_editor** (SpineDBEditor) – DB editor

msg_error

Emitted when an error occurs.

_HEADER = ('name', 'value', 'database')

_ITEM_NAME_KEY

_ITEM_VALUE_KEY

classmethod _make_adder_row(*default_db_map*)

Generates a new empty last row.

Parameters
default_db_map (*DiffDatabaseMapping*) – initial database mapping

Returns
empty row

Return type
list

abstract static _make_hidden_adder_columns()

Creates hidden extra columns for adder row.

Returns
extra columns

Return type
list

set_db_maps(*db_maps*)

Changes current database mappings.

Parameters
db_maps (*Iterable of DiffDatabaseMapping*) – database mappings

abstract _fetch_parents()

Yields fetch parents for this model.

Yields
FetchParent

canFetchMore(_)

fetchMore(_)

rowCount(*parent=QModelIndex()*)

columnCount(*parent=QModelIndex()*)

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

abstract _add_data_to_db_mngr(*name*, *value*, *db_map*)

Tells database manager to start adding data.

Parameters

- **name** (*str*) – metadata name
- **value** (*str*) – metadata value
- **db_map** (*DiffDatabaseMapping*) – database mapping

abstract _update_data_in_db_mngr(*id_*, *name*, *value*, *db_map*)

Tells database manager to start updating data.

Parameters

- **id** (*int*) – database id
- **name** (*str*) – metadata name
- **value** (*str*) – metadata value
- **db_map** (*DiffDatabaseMapping*) – database mapping

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

batch_set_data(*indexes*, *values*)

Sets data in multiple indexes simultaneously.

Parameters

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

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

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

abstract _database_table_name()

Returns primary database table name.

Returns
table name

Return type
str

abstract _row_id(*row*)

Returns a unique row id.

Parameters
row (*list*) – data table row

Returns
id or None

Return type
int

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

abstract static _ids_from_added_item(*item*)

Returns ids that uniquely identify an added database item.

Parameters
item (*dict*) – added item

Returns
unique identifier

Return type
Any

abstract static _extra_cells_from_added_item(*item*)

Constructs extra cells for data row from added database item.

Parameters
item (*dict*) – added item

Returns
extra cells

Return type
list

abstract _set_extra_columns(*row*, *ids*)

Sets extra columns for data row.

Parameters

- **row** (*list*) – data row
- **ids** (*Any*) –

_add_data(*db_map_data*)

Adds new data from database manager to the model.

Parameters
db_map_data (*dict*) – added items keyed by database mapping

_update_data(*db_map_data*, *id_column*)

Update data table after database update.

Parameters

- **db_map_data** (*dict*) – updated items keyed by database mapping
- **id_column** (*int*) – column that contains item ids

_remove_data(*db_map_data*, *id_column*)

Removes data from model after it has been removed from databases.

Parameters

- **db_map_data** (*dict*) – removed items keyed by database mapping
- **id_column** (*int*) – column that contains item ids

sort(*column*, *order=Qt.AscendingOrder*)

_find_db_map(*codename*)

Finds database mapping with given codename.

Parameters
codename (*str*) – database mapping's code name

Returns
database mapping or None if not found

Return type
DiffDatabaseMapping

_reserved_metadata()

Collects metadata names and values that are already in database.

Returns
mapping from database mapping to metadata name and value

Return type
dict

spinetoolbox.spine_db_editor.mvcmodels.mime_types

Contains mime types used by the models.

Module Contents

spinetoolbox.spine_db_editor.mvcmodels.mime_types.**ALTERNATIVE_DATA** = **'application/vnd.spinetoolbox.alternative'**

spinetoolbox.spine_db_editor.mvcmodels.mime_types.**SCENARIO_DATA** = **'application/vnd.spinetoolbox.scenario'**

spinetoolbox.spine_db_editor.mvcmodels.multi_db_tree_item

Base classes to represent items from multiple databases in a tree.

Module Contents

Classes

MultiDBTreeItem	A tree item that may belong in multiple databases.

class spinetoolbox.spine_db_editor.mvcmodels.multi_db_tree_item.**MultiDBTreeItem**(*model*, *db_map_ids=None*)

Bases: *spinetoolbox.mvcmodels.minimal_tree_model.TreeItem*

A tree item that may belong in multiple databases.

Parameters

- **model** (MinimalTreeModel, *optional*) – item's model
- **db_map_ids** (*dict, optional*) – maps instances of DatabaseMapping to the id of the item in that db

property visible_children

property db_mngr

abstract property child_item_class

Returns the type of child items.

property display_id

Returns an id for display based on the display key. This id must be the same across all db_maps. If it's not, this property becomes None and measures need to be taken (see update_children_by_id).

property name

property display_data
Returns the name for display.

property display_database
Returns the database for display.

property display_icon
Returns an icon to display next to the name. Reimplement in subclasses to return something nice.

property first_db_map
Returns the first associated db_map.

property db_maps
Returns a list of all associated db_maps.

property db_map_ids
Returns dict with db_map as key and id as value

property _children_sort_key

property fetch_item_type

item_type
Item type identifier string. Should be set to a meaningful value by subclasses.

visual_key = ['name']

_fetch_index

row_count()
Overriden to use visible_children.

child(*row*)
Overriden to use visible_children.

child_number()
Overriden to use find_row which is a dict-lookup rather than a list.index() call.

refresh_child_map()
Recomputes the child map.

abstract set_data(*column*, *value*, *role*)
Sets data for this item.

Parameters

- **column** (*int*) – column index
- **value** (*object*) – a new value
- **role** (*int*) – role of the new value

Returns
True if data was set successfully, False otherwise

Return type
bool

add_db_map_id(*db_map*, *id_*)
Adds id for this item in the given db_map.

take_db_map(*db_map*)

Removes the mapping for given db_map and returns it.

deep_refresh_children()

Refreshes children after taking db_maps from them. Called after removing and updating children for this item.

deep_remove_db_map(*db_map*)

Removes given db_map from this item and all its descendants.

deep_take_db_map(*db_map*)

Removes given db_map from this item and all its descendants, and returns a new item from the db_map's data.

Returns
MultiDBTreeItem, NoneType

deep_merge(*other*)

Merges another item and all its descendants into this one.

db_map_id(*db_map*)

Returns the id for this item in given db_map or None if not present.

db_map_data(*db_map*)

Returns data for this item in given db_map or an empty dict if not present.

db_map_data_field(*db_map*, *field*, *default=None*)

Returns field from data for this item in given db_map or None if not found.

_create_new_children(*db_map*, *children_ids*, ***kwargs*)

Creates new items from ids associated to a db map.

Parameters

- **db_map** (*DiffDatabaseMapping*) – create children for this db_map
- **children_ids** (*iter*) – create children from these ids

Returns
new children

Return type
list of MultiDBTreeItem

make_or_restore_child(*db_map*, *id_*, ***kwargs*)

Makes or restores a child if one was ever made using given db_map and id. The purpose of restoring is to keep using the same FetchParent, which is useful in case the user undoes a series of removal operations that would add items in cascade to the tree.

Parameters

- **db_map** (*DatabaseMapping*) –
- **id** (*int*) –

Returns
MultiDBTreemItem

restore(*db_map_ids*, ***kwargs*)

_make_child(*db_map_ids*, ***kwargs*)

_merge_children(*new_children*)

Merges new children into this item. Ensures that each child has a valid display id afterwards.

_insert_children_sorted(*new_children*)

Inserts and sorts children.

can_fetch_more()

Returns whether this item can fetch more.

fetch_more()

Fetches children from all associated databases.

fetch_more_if_possible()

_key_for_index(*db_map*)

accepts_item(*item*, *db_map*)

handle_items_added(*db_map_data*)

handle_items_removed(*db_map_data*)

handle_items_updated(*db_map_data*)

append_children_by_id(*db_map_ids*, ***kwargs*)

Appends children by id.

Parameters
db_map_ids (*dict*) – maps DiffDatabaseMapping instances to list of ids

remove_children_by_id(*db_map_ids*)

Removes children by id.

Parameters
db_map_ids (*dict*) – maps DiffDatabaseMapping instances to list of ids

is_valid()

See base class.

update_children_by_id(*db_map_ids*, ***kwargs*)

Updates children by id. Essentially makes sure all children have a valid display id after updating the underlying data. These may require 'splitting' a child into several for different dbs or merging two or more children from different dbs.

Examples of problems:

- The user renames an object_class in one db but not in the others –> we need to split
- The user renames an object_class and the new name is already 'taken' by another object_class in another db_map –> we need to merge

Parameters
db_map_ids (*dict*) – maps DiffDatabaseMapping instances to list of ids

insert_children(*position*, *children*)

Inserts new children at given position.

Parameters

- **position** (*int*) – insert new items here

- **children** (*Iterable of MultiDBTreeItem*) – insert items from this iterable

Returns
True if children were inserted successfully, False otherwise

Return type
bool

remove_children(*position*, *count*)

Removes count children starting from the given position.

reposition_child(*row*)

find_row(*db_map*, *id_*)

find_children_by_id(*db_map*, **ids*, *reverse=True*)

Generates children with the given ids in the given db_map. If the first id is None, then generates *all* children with the given db_map.

find_rows_by_id(*db_map*, **ids*, *reverse=True*)

_find_unsorted_rows_by_id(*db_map*, **ids*)

Generates rows corresponding to children with the given ids in the given db_map. If the only id given is None, then generates rows corresponding to *all* children with the given db_map.

data(*column*, *role=Qt.ItemDataRole.DisplayRole*)

Returns data for given column and role.

default_parameter_data()

Returns data to set as default in a parameter table when this item is selected.

tear_down()

Do stuff after the item has been removed.

register_fetch_parent()

Registers item's fetch parent for all model's databases.

spinetoolbox.spine_db_editor.mvcmodels.multi_db_tree_model

A base model class to represent items from multiple databases in a tree.

Module Contents

Classes

MultiDBTreeModel	Base class for all tree models in Spine db editor.

class spinetoolbox.spine_db_editor.mvcmodels.multi_db_tree_model.**MultiDBTreeModel**(*db_editor*, *db_mngr*, **db_maps*)

Bases: *spinetoolbox.mvcmodels.minimal_tree_model.MinimalTreeModel*

Base class for all tree models in Spine db editor.

Init class.

Parameters

- **db_editor** (SpineDBEditor) –
- **db_mngr** (SpineDBManager) – A manager for the given db_maps
- ***db_maps** – DatabaseMapping instances

abstract property root_item_type

Implement in subclasses to create a model specific to any entity type.

property root_item

property root_index

property _header_labels

build_tree()

Builds tree.

columnCount(*parent=QModelIndex()*)

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

find_items(*db_map*, *path_prefix*, *fetch=False*)

Returns items at given path prefix.

spinetoolbox.spine_db_editor.mvcmodels.parameter_value_list_item

Tree items for parameter_value lists.

Module Contents

Classes

DBItem	An item representing a db.
ListItem	A list item.
ValueItem	Paints the item gray if it's the last.

class spinetoolbox.spine_db_editor.mvcmodels.parameter_value_list_item.**DBItem**(**args*, ***kwargs*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.EmptyChildMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.FetchMoreMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.StandardDBItem*

An item representing a db.

property item_type

property fetch_item_type

empty_child()

_make_child(*id_*)

class spinetoolbox.spine_db_editor.mvcmodels.parameter_value_list_item.**ListItem**(**args*, ***kwargs*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.GrayIfLastMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.EditableMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.EmptyChildMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.SortChildrenMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.BoldTextMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.FetchMoreMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.LeafItem*

A list item.

property item_type

property fetch_item_type

_make_item_data()

_do_set_up()

Do stuff after the item has been inserted.

empty_child()

_make_child(*id_*)

accepts_item(*item*, *db_map*)

_children_sort_key(*child*)

data(*column*, *role=Qt.ItemDataRole.DisplayRole*)

Returns data for given column and role.

_make_item_to_add(*value*)

add_item_to_db(*db_item*)

update_item_in_db(*db_item*)

class spinetoolbox.spine_db_editor.mvcmodels.parameter_value_list_item.**ValueItem**(*model*, *identifier=None*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.GrayIfLastMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.EditableMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.LeafItem*

Paints the item gray if it's the last.

Parameters

- **model** (MinimalTreeModel) –
- **identifier** (*int, optional*) – item's database id

property item_type

data(*column*, *role=Qt.ItemDataRole.DisplayRole*)

Returns data for given column and role.

list_index()

_make_item_to_add(*value*)

_make_item_to_update(*_column*, *value*)

add_item_to_db(*db_item*)

update_item_in_db(*db_item*)

spinetoolbox.spine_db_editor.mvcmodels.parameter_value_list_model

A tree model for parameter_value lists.

Module Contents

Classes

ParameterValueListModel	A model to display parameter_value_list data in a tree view.

class spinetoolbox.spine_db_editor.mvcmodels.parameter_value_list_model.**ParameterValueListModel**(*db_editor*, *db_mngr*, **db_maps*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_model_base.TreeModelBase*

A model to display parameter_value_list data in a tree view.

Parameters

- **db_editor** (*SpineDBEditor*) –
- **db_mngr** (*SpineDBManager*) –
- ***db_maps** – DatabaseMapping instances

_make_db_item(*db_map*)

columnCount(*parent=QModelIndex()*)

Returns the number of columns under the given parent. Always 1.

index_name(*index*)

get_set_data_delayed(*index*)

Returns a function that ParameterValueEditor can call to set data for the given index at any later time, even if the model changes.

Parameters
index (*QModelIndex*) –

Returns
Callable

spinetoolbox.spine_db_editor.mvcmodels.pivot_model

Provides PivotModel.

Module Contents

Classes

PivotModel

class spinetoolbox.spine_db_editor.mvcmodels.pivot_model.**PivotModel**

property rows

property columns

reset_model(*data*, *top_left_headers=()*, *rows=()*, *columns=()*, *frozen=()*, *frozen_value=()*)

Resets the model.

clear_model()

update_model(*data*)

add_to_model(*data*)

Adds data to model.

Parameters
data (*dict*) – pivot model data

Returns
added row count and added column count

Return type
tuple

remove_from_model(*data*)

frozen_values(*data*)

Collects frozen values from data.

Parameters
data (*dict*) – pivot model data

Returns
frozen values

Return type
set of tuple

_check_pivot(*rows*, *columns*, *frozen*, *frozen_value*)

Checks if given pivot is valid.

Returns
error message or None if no error

Return type
str, NoneType

_index_key_getter(*indexes*)

Returns an itemgetter that always returns tuples from list of indexes

Parameters
indexes (*tuple*) –

Returns
an itemgetter

Return type
Callable

_get_unique_index_values(*indexes*)

Returns unique indexes that match the frozen condition.

Parameters
indexes (*tuple*) – indexes to match

Returns
unique indexes

Return type
list

set_pivot(*rows*, *columns*, *frozen*, *frozen_value*)

Sets pivot.

set_frozen_value(*value*)

Sets values for the frozen indexes.

Parameters
value (*tuple of str*) –

set_frozen(*frozen*)

Sets the frozen names without resetting the pivot.

Parameters
frozen (*Iterable of str*) –

get_pivoted_data(*row_mask*, *column_mask*)

Returns data for indexes in row_mask and column_mask.

Parameters

- **row_mask** (*list*) –
- **column_mask** (*list*) –

Returns
list(list)

row_key(*row*)

column_key(*column*)

spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models

Provides pivot table models for the Tabular View.

Module Contents

Classes

TopLeftHeaderItem	Base class for all 'top left pivot headers'.
TopLeftEntityHeaderItem	A top left header for an entity class.
TopLeftParameterHeaderItem	A top left header for parameter_definition.
TopLeftParameterIndexHeaderItem	A top left header for parameter index.
TopLeftAlternativeHeaderItem	A top left header for alternative.
TopLeftScenarioHeaderItem	A top left header for scenario.
TopLeftDatabaseHeaderItem	A top left header for database.
PivotTableModelBase	**param db_editor**
ParameterValuePivotTableModel	A model for the pivot table in parameter_value input type.
IndexExpansionPivotTableModel	A model for the pivot table in parameter index expansion input type.
ElementPivotTableModel	A model for the pivot table in element input type.
ScenarioAlternativePivotTableModel	A model for the pivot table in scenario alternative input type.
PivotTableSortFilterProxy	Initialize class.

Functions

_make_get_id(action)	Returns a function to compute the db_map-id tuple of an item.

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.**TopLeftHeaderItem**(*model*)

Base class for all 'top left pivot headers'. Represents a header located in the top left area of the pivot table.

Parameters
model (PivotTableModelBase) –

property model

property db_mngr

_get_header_data_from_db(*item_type*, *header_id*, *field_name*, *role*)

static accepts(*header_id*)

Tests if header id is valid.

Parameters
header_id (*Any*) – header id

Returns
True if id is valid, False otherwise

Return type
bool

abstract header_data(*header_id*, *role=Qt.ItemDataRole.DisplayRole*)

Returns header data for given id.

Parameters

- **header_id** (*Any*) – header id
- **role** (*Qt.ItemDataRole*) – data role

Returns
data corresponding to role

Return type
Any

abstract update_data(*db_map_data*)

Updates database data.

Parameters
db_map_data (*dict*) – update data

Returns
True if data was successfully updated, False otherwise

Return type
bool

abstract add_data(*names*, *db_map*)

Adds more data to database.

Parameters

- **names** (*set of str*) – header names
- **db_map** (*DatabaseMapping*) – database to add the data to

Returns
True if data was added successfully, False otherwise

Return type
bool

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.**TopLeftEntityHeaderItem**(*model*, *rank*, *class_name*, *class_id*)

Bases: *TopLeftHeaderItem*

A top left header for an entity class.

Parameters
model (PivotTableModelBase) –

property header_type

property name

property class_id

header_data(*header_id*, *role=Qt.ItemDataRole.DisplayRole*)

See base class.

update_data(*db_map_data*)

See base class.

add_data(*names*, *db_map*)

See base class.

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.TopLeftParameterHeaderItem(*model*)

Bases: *TopLeftHeaderItem*

A top left header for parameter_definition.

Parameters
model (PivotTableModelBase) –

property header_type

property name

header_data(*header_id*, *role=Qt.ItemDataRole.DisplayRole*)

See base class.

update_data(*db_map_data*)

See base class.

add_data(*names*, *db_map*)

See base class.

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.TopLeftParameterIndexHeaderItem(*model*)

Bases: *TopLeftHeaderItem*

A top left header for parameter index.

Parameters
model (PivotTableModelBase) –

property header_type

property name

header_data(*header_id*, *role=Qt.ItemDataRole.DisplayRole*)

See base class.

update_data(*db_map_data*)

See base class.

add_data(*names*, *db_map*)

See base class.

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.TopLeftAlternativeHeaderItem(*model*)

Bases: *TopLeftHeaderItem*

A top left header for alternative.

Parameters
model (PivotTableModelBase) –

property header_type

property name

header_data(*header_id*, *role=Qt.ItemDataRole.DisplayRole*)

See base class.

update_data(*db_map_data*)

See base class.

add_data(*names*, *db_map*)

See base class.

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.TopLeftScenarioHeaderItem(*model*)

Bases: *TopLeftHeaderItem*

A top left header for scenario.

Parameters
model (PivotTableModelBase) –

property header_type

property name

header_data(*header_id*, *role=Qt.ItemDataRole.DisplayRole*)

See base class.

update_data(*db_map_data*)

See base class.

add_data(*names*, *db_map*)

See base class.

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.TopLeftDatabaseHeaderItem(*model*)

Bases: *TopLeftHeaderItem*

A top left header for database.

Parameters
model (PivotTableModelBase) –

property header_type

property name

header_data(*header_id*, *role=Qt.ItemDataRole.DisplayRole*)

See base class.

update_data(*db_map_data*)

See base class.

add_data(*names*, *db_map*)

See base class.

set_data(*codename*)

Sets database mapping's codename.

Parameters
codename (*str*) – database codename

Returns
True if codename was acceptable, False otherwise

Return type
bool

`take_suggested_db_map()`

Suggests database mapping resetting the suggestion afterwards.

Returns
database mapping

Return type
DatabaseMapping

`suggest_db_map_codename()`

Suggests a database mapping codename.

Returns
codename

Return type
str

`class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.PivotTableModelBase(db_editor)`

Bases: `PySide6.QtCore.QAbstractTableModel`

Parameters
db_editor (SpineDBEditor) –

`property db_maps`

`abstract property item_type`

Returns the item type.

`property plot_x_column`

Returns the index of the column designated as Y values for plotting or None.

`_CHUNK_SIZE = 1000`

`model_data_changed`

`frozen_values_added`

`frozen_values_removed`

`reset_fetch_parents()`

`set_fetch_parents_non_obsolete()`

`abstract _fetch_parents()`

Yields fetch parents for this model.

Yields
FetchParent

`canFetchMore(_)`

`fetchMore(_)`

`_reset_data_count()`

_collect_more_data()

_collect_more_rows()

_collect_more_columns()

abstract call_reset_model(*pivot=None*)

Parameters
pivot (*tuple, optional*) – list of rows, list of columns, list of frozen indexes, frozen value

abstract static make_delegate(*parent*)

reset_model(*data*, *index_ids*, *rows=()*, *columns=()*, *frozen=()*, *frozen_value=()*)

clear_model()

update_model(*data*)

Update model with new data, but doesn't grow the model.

Parameters
data (*dict*) –

add_to_model(*db_map_data*)

remove_from_model(*data*)

_emit_all_data_changed()

set_pivot(*rows*, *columns*, *frozen*, *frozen_value*)

set_frozen(*frozen*)

Sets the order of frozen headers without changing model data.

Parameters
frozen (*list of str*) – new frozen

set_frozen_value(*frozen_value*)

Sets frozen value resetting the model.

Parameters
frozen_value (*tuple*) – frozen value

Returns
True if value was set, False otherwise

Return type
bool

set_plot_x_column(*column*, *is_x*)

Sets or clears the X flag on a column

x_value(*index*)

Returns x value for given model index.

Parameters
index (*QModelIndex*) – model index

Returns
x value

Return type
Any

x_parameter_name()
Returns x column's parameter name.

Returns
parameter name

Return type
str

headerRowCount()
Returns number of rows occupied by header.

headerColumnCount()
Returns number of columns occupied by header.

dataRowCount()
Returns number of rows that contain actual data.

dataColumnCount()
Returns number of columns that contain actual data.

emptyRowCount()

emptyColumnCount()

rowCount(*parent=QModelIndex()*)
Number of rows in table, number of header rows + datarows + 1 empty row

columnCount(*parent=QModelIndex()*)
Number of columns in table, number of header columns + datacolumns + 1 empty columns

flags(*index*)
Roles for data

top_left_indexes()
Returns indexes in the top left area.

Returns
list(QModelIndex): top indexes (horizontal headers, associated to rows) list(QModelIndex): left indexes (vertical headers, associated to columns)

index_within_top_left(*index*)

index_in_top(*index*)

index_in_left(*index*)

index_in_top_left(*index*)
Returns whether the given index is in top left corner, where pivot names are displayed.

index_in_column_headers(*index*)
Returns whether the given index is in column headers (horizontal) area.

index_in_row_headers(*index*)
Returns whether the given index is in row headers (vertical) area.

index_in_headers(*index*)

index_in_empty_column_headers(*index*)

Returns whether the given index is in empty column headers (vertical) area.

index_in_empty_row_headers(*index*)

Returns whether the given index is in empty row headers (vertical) area.

index_in_data(*index*)

Returns whether the given index is in data area.

column_is_index_column(*column*)

Returns True if column is the column containing expanded parameter_value indexes.

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

map_to_pivot(*index*)

Returns a tuple of row and column in the pivot model that corresponds to the given model index.

Parameters
index (*QModelIndex*) –

Returns
row int: column

Return type
int

top_left_id(*index*)

Returns the id of the top left header corresponding to the given header index.

Parameters
index (*QModelIndex*) –

Returns
int, NoneType

_header_id(*index*)

Returns the id of the given row or column header index.

Parameters
index (*QModelIndex*) –

Returns
tuple or DatabaseMapping or NoneType

_header_ids(*row*, *column*)

Returns the ids for the headers at given row *and* column.

Parameters

- **row** (*int*) –
- **column** (*int*) –

Returns
tuple(int)

header_name(*index*)

Returns the name corresponding to the given header index. Used by PivotTableView.

Parameters
index (*QModelIndex*) –

Returns
str

_color_data(*index*)

_text_alignment_data(*index*)

_header_data(*index*)

_header_name(*top_left_id*, *header_id*)

get_db_map_entities()
Returns a dict mapping db maps to a list of dict entity items in the current class.

Returns
dict

abstract _data(*index*, *role*)

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

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

batch_set_data(*indexes*, *values*)

_batch_set_inner_data(*inner_data*)

abstract _do_batch_set_inner_data(*row_map*, *column_map*, *data*, *values*)

_batch_set_header_data(*header_data*)
Sets header data for multiple indexes at once.

Parameters
header_data (*list of tuple*) – mapping from index to data

Returns
True if data was set successfully, False otherwise

Return type
bool

_batch_set_empty_header_data(*header_data*, *get_top_left_id*)

tear_down()
Sets fetch parents obsolete preventing further updates.

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.**ParameterValuePivotTableModel**(*parent*)
Bases: *PivotTableModelBase*

A model for the pivot table in parameter_value input type.

Parameters
parent (SpineDBEditor) –

property item_type
Returns the item type.

_handle_entity_classes_added(*db_map_data*)

_handle_entity_classes_removed(*db_map_data*)

_handle_entities_added(*db_map_data*)

_handle_entities_removed(*db_map_data*)

_handle_parameter_definitions_added(*db_map_data*)

_handle_parameter_definitions_removed(*db_map_data*)

_handle_parameter_values_added(*db_map_data*)

_handle_parameter_values_removed(*db_map_data*)

_handle_alternatives_added(*db_map_data*)

_handle_alternatives_removed(*db_map_data*)

_load_empty_parameter_value_data(*db_map_entities=None*, *db_map_parameter_ids=None*, *db_map_alternative_ids=None*)

Returns a dict containing all possible combinations of entities and parameters for the current class in all db_maps.

Parameters

- **db_map_entities** (*dict, optional*) – if given, only load data for these db maps and entities
- **db_map_parameter_ids** (*dict, optional*) – if given, only load data for these db maps and parameter definitions
- **db_map_alternative_ids** (*dict, optional*) – if given, only load data for these db maps and alternatives

Returns
Key is a tuple object_id, ..., parameter_id, value is None.

Return type
dict

_all_combination_for_empty_parameter_value(*db_map_entities*, *db_map_parameter_ids*, *db_map_alternative_ids*)

_load_full_parameter_value_data(*db_map_parameter_values=None*, *action='add'*)

Returns a dict of parameter values for the current class.

Parameters

- **db_map_parameter_values** (*list, optional*) –
- **action** (*str*) –

Returns
Key is a tuple object_id, ..., parameter_id, value is the parameter_value.

Return type
dict

_fetch_parents()

Yields fetch parents for this model.

Yields
FetchParent

_db_map_element_ids(*header_ids*)

db_map_entity_ids(*indexes*)

Returns db_map and entity ids for given indexes. Used by PivotTableView.

Parameters
indexes (*list of QModelIndex*) – indexes corresponding to entity items

Returns
mapping DatabaseMapping to set of entity ids

Return type
dict

all_header_names(*index*)

Returns the entity, parameter, alternative, and db names corresponding to the given data index.

Parameters
index (*QModelIndex*) –

Returns
object names str: parameter name str: alternative name str: db name

Return type
list(str)

index_name(*index*)

Returns a string that concatenates the object and parameter names corresponding to the given data index. Used by plotting and ParameterValueEditor.

Parameters
index (*QModelIndex*) –

Returns
str

column_name(*column*)

Returns a string that concatenates the object and parameter names corresponding to the given column. Used by plotting.

Parameters
column (*int*) –

Returns
str

call_reset_model(*pivot=None*)

See base class.

static make_delegate(*parent*)

_default_pivot(*data*)

_data(*index*, *role*)

_do_batch_set_inner_data(*row_map*, *column_map*, *data*, *values*)

_entity_parameter_value_to_add(*db_map*, *header_ids*, *value_and_type*, *ent_id_lookup=None*)

_make_parameter_value_to_add()

static **_parameter_value_to_update**(*id_*, *header_ids*, *value_and_type*)

_batch_set_parameter_value_data(*row_map*, *column_map*, *data*, *values*)

Sets parameter values in batch.

_add_parameter_values(*db_map_data*)

_update_parameter_values(*db_map_data*)

get_set_data_delayed(*index*)

Returns a function that ParameterValueEditor can call to set data for the given index at any later time, even if the model changes.

Parameters
index (*QModelIndex*) –

Returns
function

_get_db_map_parameter_value_or_def_ids(*item_type*)

Returns a dict mapping db maps to a list of integer parameter (value or def) ids from the current class.

Parameters
item_type (*str*) – either "parameter_value" or "parameter_definition"

Returns
dict

_get_db_map_parameter_values_or_defs(*item_type*)

Returns a dict mapping db maps to list of dict parameter (value or def) items from the current class.

Parameters
item_type (*str*) – either "parameter_value" or "parameter_definition"

Returns
dict

load_full_parameter_value_data(*db_map_parameter_values=None*, *action='add'*)

Returns a dict of parameter values for the current class.

Parameters

- **db_map_parameter_values** (*list, optional*) –
- **action** (*str*) –

Returns
Key is a tuple object_id, . . . , parameter_id, value is the parameter_value.

Return type
dict

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.**IndexExpansionPivotTableModel**(*parent*)

Bases: *ParameterValuePivotTableModel*

A model for the pivot table in parameter index expansion input type.

Parameters
parent (SpineDBEditor) –

INDEX_INSERTION_POINT

call_reset_model(*pivot=None*)

See base class.

emptyRowCount()

emptyColumnCount()

flags(*index*)

Roles for data

column_is_index_column(*column*)

Returns True if column is the column containing expanded parameter_value indexes.

_handle_parameter_values_removed(*db_map_data*)

_load_empty_parameter_value_data(*db_map_entities=None*, *db_map_parameter_ids=None*, *db_map_alternative_ids=None*)

Does not load the data since adding values in index expansion mode is disabled.

Parameters

- **db_map_entities** (*dict, optional*) – mapping from database map to iterable of entity items
- **db_map_parameter_ids** (*dict, optional*) – mapping from database map to iterable of parameter definition id tuples
- **db_map_alternative_ids** (*dict, optional*) – mapping from database map to iterable of alternative id tuples

Returns
empty data

Return type
dict

_load_full_parameter_value_data(*db_map_parameter_values=None*, *action='add'*)

Makes a dict of expanded parameter values for the current class.

Parameters

- **db_map_parameter_values** (*list, optional*) –
- **action** (*str*) –

Returns
mapping from unique value id tuple to value tuple

Return type
dict

_data(*index*, *role*)

static _parameter_value_to_update(*id_*, *header_ids*, *value_and_type*)

_update_parameter_values(*db_map_data*)

_indexes(*value*)

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.**ElementPivotTableModel**(*parent*)

Bases: *PivotTableModelBase*

A model for the pivot table in element input type.

Parameters
parent (SpineDBEditor) –

property item_type

Returns the item type.

_handle_entities_added(*db_map_data*)

_handle_entities_removed(*db_map_data*)

_load_empty_element_data(*db_map_data*)

_handle_elements_added(*db_map_data*)

_handle_elements_removed(*db_map_data*)

_fetch_parents()

Yields fetch parents for this model.

Yields
FetchParent

call_reset_model(*pivot=None*)

See base class.

static make_delegate(*parent*)

_default_pivot(*data*)

_data(*index*, *role*)

_do_batch_set_inner_data(*row_map*, *column_map*, *data*, *values*)

_batch_set_entity_data(*row_map*, *column_map*, *data*, *values*)

_load_full_element_data(*db_map_entities=None*, *action='add'*)

Returns a dict of entity elements in the current class.

Parameters

- **db_map_entities** (*dict, optional*) – a mapping from database map to entities in the current entity class
- **action** (*str*) – 'add' or 'remove'

Returns
Key is db_map-object id tuple, value is relationship id.

Return type
dict

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.**ScenarioAlternativePivotTableModel**(*paren*

Bases: *PivotTableModelBase*

A model for the pivot table in scenario alternative input type.

Parameters
parent (SpineDBEditor) –

property item_type

Returns the item type.

_handle_scenarios_added(*db_map_data*)

_handle_scenarios_removed(*db_map_data*)

_handle_alternatives_added(*db_map_data*)

_handle_alternatives_removed(*db_map_data*)

_handle_scenario_alternatives_changed(*db_map_data*)

_fetch_parents()

Yields fetch parents for this model.

Yields
FetchParent

call_reset_model(*pivot=None*)

See base class.

static make_delegate(*parent*)

_default_pivot(*data*)

_data(*index*, *role*)

_do_batch_set_inner_data(*row_map*, *column_map*, *data*, *values*)

_batch_set_scenario_alternative_data(*row_map*, *column_map*, *data*, *values*)

_load_scenario_alternative_data(*db_map_scenarios=None*, *db_map_alternatives=None*)

Returns a dict containing all scenario alternatives.

Returns
Key is db_map-id tuple, value is None or rank.

Return type
dict

class spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.**PivotTableSortFilterProxy**(*parent=None*)

Bases: PySide6.QtCore.QSortFilterProxyModel

Initialize class.

model_data_changed

setSourceModel(*model*)

set_filter(*identifier*, *filter_value*)

Sets filter for a given index (object_class) name.

Parameters

- **identifier** (*int*) – index identifier
- **filter_value** (*set, None*) – A set of accepted values, or None if no filter (all pass)

clear_filter()

accept_index(*index*, *index_ids*)

filterAcceptsRow(*source_row*, *source_parent*)

Returns true if the item in the row indicated by the given source_row and source_parent should be included in the model; otherwise returns false.

filterAcceptsColumn(*source_column*, *source_parent*)

Returns true if the item in the column indicated by the given source_column and source_parent should be included in the model; otherwise returns false.

batch_set_data(*indexes*, *values*)

spinetoolbox.spine_db_editor.mvcmodels.pivot_table_models.**_make_get_id**(*action*)

Returns a function to compute the db_map-id tuple of an item.

Parameters
action (*str*) – "add" or "remove"

Returns
function to compute db_map-id tuples

Return type
Callable

spinetoolbox.spine_db_editor.mvcmodels.scenario_item

Classes to represent items in scenario tree.

Module Contents

Classes

ScenarioDBItem	A root item representing a db.
ScenarioItem	A scenario leaf item.
ScenarioAlternativeItem	A scenario alternative leaf item.

Attributes

_SCENARIO_ICON

spinetoolbox.spine_db_editor.mvcmodels.scenario_item.**_SCENARIO_ICON = '\uf008'**

class spinetoolbox.spine_db_editor.mvcmodels.scenario_item.**ScenarioDBItem**(**args*, ***kwargs*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.EmptyChildMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.FetchMoreMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.StandardDBItem*

A root item representing a db.

property item_type

property fetch_item_type

empty_child()

_make_child(*id_*)

class spinetoolbox.spine_db_editor.mvcmodels.scenario_item.**ScenarioItem**(**args*, ***kwargs*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.GrayIfLastMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.EditableMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.EmptyChildMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.FetchMoreMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.BoldTextMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.LeafItem*

A scenario leaf item.

property item_type

property fetch_item_type

property icon_code

property alternative_id_list

tool_tip(*column*)

_do_set_up()

Doesn't add children to the last row.

add_item_to_db(*db_item*)

update_item_in_db(*db_item*)

handle_updated_in_db()

flags(*column*)

Makes items editable.

update_alternative_id_list()

accepts_item(*item*, *db_map*)

handle_items_added(*_db_map_data*)

Inserts items at right positions. Items with commit_id are kept sorted. Items without a commit_id are put at the end.

Parameters

db_map_data (*dict*) – mapping db_map to list of dict corresponding to db items

handle_items_removed(*_db_map_data*)

handle_items_updated(*_db_map_data*)

empty_child()

See base class.

_make_child(*id_*)

Not needed - we don't quite add children here, but rather update them in update_alternative_id_list.

class spinetoolbox.spine_db_editor.mvcmodels.scenario_item.**ScenarioAlternativeItem**(*model*, *identifier=None*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.GrayIfLastMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.EditableMixin*, *spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.LeafItem*

A scenario alternative leaf item.

Parameters

- **model** (MinimalTreeModel) –
- **identifier** (*int, optional*) – item's database id

property item_type

property item_data

property alternative_id

tool_tip(*column*)

_make_item_data()

abstract add_item_to_db(*db_item*)

abstract update_item_in_db(*db_item*)

flags(*column*)

Makes items editable.

set_data(*column*, *value*, *role=Qt.ItemDataRole.EditRole*)

Sets data for this item.

Parameters

- **column** (*int*) – column index
- **value** (*object*) – a new value
- **role** (*int*) – role of the new value

Returns

True if data was set successfully, False otherwise

Return type

bool

spinetoolbox.spine_db_editor.mvcmodels.scenario_model

Contains scenario tree model.

Module Contents

Classes

ScenarioModel	A model to display scenarios in a tree view.

class spinetoolbox.spine_db_editor.mvcmodels.scenario_model.**ScenarioModel**(*db_editor*, *db_mngr*, **db_maps*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.tree_model_base.TreeModelBase*

A model to display scenarios in a tree view.

Parameters

- **db_editor** (SpineDBEditor) –
- **db_mngr** (SpineDBManager) –
- ***db_maps** – DatabaseMapping instances

_make_db_item(*db_map*)

supportedDropActions()

mimeData(*indexes*)

Stores selected indexes into MIME data.

If indexes contains scenario indexes, only those indexes will be kept. Otherwise, only scenario alternative indexes are kept.

The MIME data contains distinct data: - Text representation of the selection - A pickled dict mapping db identifier to list of alternative ids - A pickled dict mapping db identifier to list of scenario ids

Parameters
indexes (*Sequence of QModelIndex*) – selected indexes

Returns
MIME data or None if selection was bad

Return type
QMimeData

canDropMimeData(*mime_data*, *drop_action*, *row*, *column*, *parent*)

dropMimeData(*mime_data*, *drop_action*, *row*, *column*, *parent*)

paste_alternative_mime_data(*mime_data*, *row*, *scenario_item*)

Adds alternatives from MIME data to the model.

Parameters

- **mime_data** (*QMimeData*) – mime data that must contain ALTERNATIVE_DATA format
- **row** (*int*) – where to paste within scenario item, -1 lets the model choose
- **scenario_item** (ScenarioItem) – parent item

paste_scenario_mime_data(*mime_data*, *db_item*)

Adds scenarios and their alternatives from MIME data to the model.

Parameters

- **mime_data** (*QMimeData*) – mime data that must contain ALTERNATIVE_DATA format
- **db_item** (ScenarioDBItem) – parent item

duplicate_scenario(*scenario_item*)

Duplicates scenario within database.

Parameters
scenario_item (ScenarioItem) – scenario item to duplicate

spinetoolbox.spine_db_editor.mvcmodels.single_and_empty_model_mixins

Miscellaneous mixins for parameter models.

Module Contents

Classes

ConvertToDBMixin	Base class for all mixins that convert model items (name-based) into database items (id-based).
SplitValueAndTypeMixin	Base class for all mixins that convert model items (name-based) into database items (id-based).
MakeEntityOnTheFlyMixin	Makes relationships on the fly.

class spinetoolbox.spine_db_editor.mvcmodels.single_and_empty_model_mixins.**ConvertToDBMixin**

Base class for all mixins that convert model items (name-based) into database items (id-based).

_convert_to_db(*item*)

Returns a db item (id-based) from the given model item (name-based).

Parameters
item (*dict*) – the model item

Returns
the db item list: error log

Return type
dict

class spinetoolbox.spine_db_editor.mvcmodels.single_and_empty_model_mixins.**SplitValueAndTypeMixin**

Bases: *ConvertToDBMixin*

Base class for all mixins that convert model items (name-based) into database items (id-based).

_convert_to_db(*item*)

Returns a db item (id-based) from the given model item (name-based).

Parameters
item (*dict*) – the model item

Returns
the db item list: error log

Return type
dict

class spinetoolbox.spine_db_editor.mvcmodels.single_and_empty_model_mixins.**MakeEntityOnTheFlyMixin**

Bases: *ConvertToDBMixin*

Makes relationships on the fly.

static _make_entity_on_the_fly(*item*, *db_map*)

Returns a database entity item (id-based) from the given model parameter_value item (name-based).

Parameters

- **item** (*dict*) – the model parameter_value item
- **db_map** (*DiffDatabaseMapping*) – the database where the resulting item belongs

Returns
the db entity item list: error log

Return type
dict

spinetoolbox.spine_db_editor.mvcmodels.single_models

Single models for parameter definitions and values (as 'for a single entity').

Module Contents

Classes

HalfSortedTableModel	Table model for outlining simple tabular data.
SingleModelBase	Base class for all single models that go in a CompoundModelBase subclass.
FilterEntityAlternativeMixin	Provides the interface to filter by entity and alternative.
ParameterMixin	Provides the data method for parameter values and definitions.
EntityMixin	

SingleParameterDefinitionModel	A parameter_definition model for a single entity_class.
SingleParameterValueModel	A parameter_value model for a single entity_class.
SingleEntityAlternativeModel	An entity_alternative model for a single entity_class.

class spinetoolbox.spine_db_editor.mvcmodels.single_models.**HalfSortedTableModel**(*parent=None*, *header=None*, *lazy=True*)

Bases: *spinetoolbox.mvcmodels.minimal_table_model.MinimalTableModel*

Table model for outlining simple tabular data.

Parameters

- **parent** (*QObject, optional*) – the parent object
- **header** (*list of str*) – header labels
- **lazy** (*boolean*) – if True, fetches data lazily

reset_model(*main_data=None*)

Reset model.

add_rows(*data*)

_sort_key(*element*)

class spinetoolbox.spine_db_editor.mvcmodels.single_models.**SingleModelBase**(*parent*, *db_map*, *entity_class_id*, *committed*, *lazy=False*)

Bases: *HalfSortedTableModel*

Base class for all single models that go in a CompoundModelBase subclass.

Parameters

- **parent** (*CompoundModelBase*) – the parent model
- **db_map** (*DatabaseMapping*) –
- **entity_class_id** (*int*) –
- **committed** (*bool*) –

abstract property item_type

The DB item type, required by the data method.

property field_map

abstract property _references

property entity_class_name

property dimension_id_list

property fixed_fields

property group_fields

property can_be_filtered

__lt__(*other*)

update_items_in_db(*items*)

Update items in db. Required by batch_set_data

_mapped_field(*field*)

item_id(*row*)

Returns parameter id for row.

Parameters
row (*int*) – row index

Returns
parameter id

Return type
int

`item_ids`()

Returns model's parameter ids.

Returns
ids

Return type
set of int

`db_item`(*index*)

`_db_item`(*row*)

`db_item_from_id`(*id_*)

`db_items`()

`flags`(*index*)

Make fixed indexes non-editable.

`_filter_accepts_row`(*row*)

`filter_accepts_item`(*item*)

`set_auto_filter`(*field*, *values*)

`_auto_filter_accepts_item`(*item*)

Returns the result of the auto filter.

`accepted_rows`()

Yields accepted rows, for convenience.

`_get_ref`(*db_item*, *field*)

Returns the item referred by the given field.

`data`(*index*, *role=Qt.ItemDataRole.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.

`batch_set_data`(*indexes*, *data*)

Sets data for indexes in batch. Sets data directly in database using db mngr. If successful, updated data will be automatically seen by the data method.

`class` spinetoolbox.spine_db_editor.mvcmodels.single_models.`FilterEntityAlternativeMixin`(**args*, ***kwargs*)

Provides the interface to filter by entity and alternative.

`set_filter_entity_ids`(*db_map_class_entity_ids*)

`set_filter_alternative_ids`(*db_map_alternative_ids*)

filter_accepts_item(*item*)

Reimplemented to also account for the entity and alternative filter.

_entity_filter_accepts_item(*item*)

Returns the result of the entity filter.

_alternative_filter_accepts_item(*item*)

Returns the result of the alternative filter.

class spinetoolbox.spine_db_editor.mvcmodels.single_models.**ParameterMixin**

Provides the data method for parameter values and definitions.

property value_field

property parameter_definition_id_key

property _references

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

Gets the id and database for the row, and reads data from the db manager using the item_type property. Paint the object_class icon next to the name. Also paint background of fixed indexes gray and apply custom format to JSON fields.

class spinetoolbox.spine_db_editor.mvcmodels.single_models.**EntityMixin**

update_items_in_db(*items*)

Overriden to create entities on the fly first.

abstract _do_update_items_in_db(*db_map_data*)

class spinetoolbox.spine_db_editor.mvcmodels.single_models.**SingleParameterDefinitionModel**(*parent*, *db_map*, *entity_class_id*, *committed*, *lazy=False*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.single_and_empty_model_mixins.SplitValueAndTypeMixin*, *ParameterMixin*, *SingleModelBase*

A parameter_definition model for a single entity_class.

Parameters

- **parent** (*CompoundModelBase*) – the parent model
- **db_map** (*DatabaseMapping*) –
- **entity_class_id** (*int*) –
- **committed** (*bool*) –

property item_type

The DB item type, required by the data method.

_sort_key(*element*)

_do_update_items_in_db(*db_map_data*)

class spinetoolbox.spine_db_editor.mvcmodels.single_models.**SingleParameterValueModel**(**args*, ***kwargs*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.single_and_empty_model_mixins.MakeEntityOnTheFlyMixin*, *spinetoolbox.spine_db_editor.mvcmodels.single_and_empty_model_mixins.SplitValueAndTypeMixin*, *ParameterMixin*, *EntityMixin*, *FilterEntityAlternativeMixin*, *SingleModelBase*

A parameter_value model for a single entity_class.

property item_type

The DB item type, required by the data method.

_sort_key(*element*)

_do_update_items_in_db(*db_map_data*)

class spinetoolbox.spine_db_editor.mvcmodels.single_models.**SingleEntityAlternativeModel**(**args*, ***kwargs*)

Bases: *spinetoolbox.spine_db_editor.mvcmodels.single_and_empty_model_mixins.MakeEntityOnTheFlyMixin*, *EntityMixin*, *FilterEntityAlternativeMixin*, *SingleModelBase*

An entity_alternative model for a single entity_class.

property item_type

The DB item type, required by the data method.

property _references

_sort_key(*element*)

_do_update_items_in_db(*db_map_data*)

spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility

A tree model for parameter_value lists.

Module Contents

Classes

StandardTreeItem	A tree item that fetches their children as they are inserted.
EditableMixin	
GrayIfLastMixin	Paints the item gray if it's the last.
BoldTextMixin	Bolds text.
EmptyChildMixin	Guarantees there's always an empty child.
SortChildrenMixin	
FetchMoreMixin	
StandardDBItem	An item representing a db.
LeafItem	A tree item that fetches their children as they are inserted.

class spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.**StandardTreeItem**(*model*)

Bases: *spinetoolbox.mvcmodels.minimal_tree_model.TreeItem*

A tree item that fetches their children as they are inserted.

Parameters
model (MinimalTreeModel) – The model where the item belongs.

property item_type

property db_mngr

property display_data

property icon_code

property display_icon

property non_empty_children

property children_ids

tool_tip(*column*)

data(*column*, *role=Qt.ItemDataRole.DisplayRole*)

Returns data for given column and role.

set_data(*column*, *value*, *role=Qt.ItemDataRole.DisplayRole*)

Sets data for this item.

Parameters

- **column** (*int*) – column index
- **value** (*object*) – a new value
- **role** (*int*) – role of the new value

Returns
True if data was set successfully, False otherwise

Return type
bool

class spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.**EditableMixin**

flags(*column*)

Makes items editable.

class spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.**GrayIfLastMixin**

Paints the item gray if it's the last.

data(*column*, *role=Qt.ItemDataRole.DisplayRole*)

class spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.**BoldTextMixin**

Bolds text.

data(*column*, *role=Qt.ItemDataRole.DisplayRole*)

class spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.**EmptyChildMixin**

Guarantees there's always an empty child.

property non_empty_children

abstract empty_child()

_do_set_up()

class spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.SortChildrenMixin

_children_sort_key(*child*)

insert_children_sorted(*children*)

class spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.FetchMoreMixin(**args*, ***kwargs*)

property fetch_item_type

tear_down()

_fetch_parents()

can_fetch_more()

fetch_more()

abstract _make_child(*id_*)

_do_make_child(*id_*)

accepts_item(*item*, *db_map*)

handle_items_added(*db_map_data*)

Inserts items at right positions. Items with commit_id are kept sorted. Items without a commit_id are put at the end.

Parameters
db_map_data (*dict*) – mapping db_map to list of dict corresponding to db items

handle_items_removed(*db_map_data*)

handle_items_updated(*db_map_data*)

class spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.StandardDBItem(*model*, *db_map*)

Bases: *SortChildrenMixin*, *StandardTreeItem*

An item representing a db.

Init class.

Parameters

- **model** (MinimalTreeModel) –
- **db_map** (*DatabaseMapping*) –

property item_type

data(*column*, *role=Qt.ItemDataRole.DisplayRole*)

Shows Spine icon for fun.

class spinetoolbox.spine_db_editor.mvcmodels.tree_item_utility.**LeafItem**(*model*, *identifier=None*)

Bases: *StandardTreeItem*

A tree item that fetches their children as they are inserted.

Parameters

- **model** (MinimalTreeModel) –
- **identifier** (*int, optional*) – item's database id

abstract property item_type

property db_map

property id

property item_data

property name

_make_item_data()

tool_tip(*column*)

abstract add_item_to_db(*db_item*)

abstract update_item_in_db(*db_item*)

header_data(*column*)

data(*column*, *role=Qt.ItemDataRole.DisplayRole*)

Returns data for given column and role.

set_data(*column*, *value*, *role=Qt.ItemDataRole.EditRole*)

Sets data for this item.

Parameters

- **column** (*int*) – column index
- **value** (*object*) – a new value
- **role** (*int*) – role of the new value

Returns

True if data was set successfully, False otherwise

Return type

bool

_make_item_to_add(*value*)

_make_item_to_update(*column*, *value*)

handle_updated_in_db()

can_fetch_more()

Returns whether this item can fetch more.

spinetoolbox.spine_db_editor.mvcmodels.tree_model_base

Models to represent things in a tree.

Module Contents

Classes

TreeModelBase	A base model to display items in a tree view.

class spinetoolbox.spine_db_editor.mvcmodels.tree_model_base.**TreeModelBase**(*db_editor*, *db_mngr*, **db_maps*)

Bases: *spinetoolbox.mvcmodels.minimal_tree_model.MinimalTreeModel*

A base model to display items in a tree view.

Parameters

- **db_editor** (SpineDBEditor) –
- **db_mngr** (SpineDBManager) –
- ***db_maps** – DatabaseMapping instances

columnCount(*parent=QModelIndex()*)

Returns the number of columns under the given parent. Always 2.

Returns
column count

Return type
int

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

build_tree()

Builds tree.

abstract _make_db_item(*db_map*)

static db_item(*item*)

db_row(*item*)

_tear_down_tree(*obj=None*)

Tears down tree items recursively

spinetoolbox.spine_db_editor.mvcmodels.utils

General helper functions and classes for DB editor's models.

Module Contents

Functions

two_column_as_csv(indexes)	Writes data in given indexes into a CSV table.

spinetoolbox.spine_db_editor.mvcmodels.utils.**two_column_as_csv**(*indexes*)

Writes data in given indexes into a CSV table.

Expects the source table to have two columns.

Parameters
indexes (*Sequence of QModelIndex*) – model indexes

Returns
data as CSV table

Return type
str

spinetoolbox.spine_db_editor.ui

Automatically generated UI modules for Spine db editor.

Submodules

spinetoolbox.spine_db_editor.ui.commit_viewer_affected_item_info

Module Contents

Classes

Ui_Form	

class spinetoolbox.spine_db_editor.ui.commit_viewer_affected_item_info.**Ui_Form**

Bases: object

setupUi(*Form*)

retranslateUi(*Form*)

spinetoolbox.spine_db_editor.ui.db_commit_viewer

Module Contents

Classes

Ui_DBCommitViewer

class spinetoolbox.spine_db_editor.ui.db_commit_viewer.**Ui_DBCommitViewer**

Bases: object

setupUi(*DBCommitViewer*)

retranslateUi(*DBCommitViewer*)

spinetoolbox.spine_db_editor.ui.scenario_generator

Module Contents

Classes

Ui_Form

class spinetoolbox.spine_db_editor.ui.scenario_generator.**Ui_Form**

Bases: object

setupUi(*Form*)

retranslateUi(*Form*)

spinetoolbox.spine_db_editor.ui.select_databases

Module Contents

Classes

Ui_Form

class spinetoolbox.spine_db_editor.ui.select_databases.**Ui_Form**

Bases: object

setupUi(*Form*)

retranslateUi(*Form*)

spinetoolbox.spine_db_editor.ui.spine_db_editor_window

Module Contents

Classes

Ui_MainWindow

class spinetoolbox.spine_db_editor.ui.spine_db_editor_window.**Ui_MainWindow**

Bases: `object`

setupUi(*MainWindow*)

retranslateUi(*MainWindow*)

spinetoolbox.spine_db_editor.widgets

Interface logic for Spine db editor.

Submodules

spinetoolbox.spine_db_editor.widgets.add_items_dialogs

Classes for custom QDialogs to add items to databases.

Module Contents

Classes

AddReadyEntitiesDialog	A dialog to let the user add new 'ready' multidimensional entities.
AddItemsDialog	A dialog to query user's preferences for new db items.
AddEntityClassesDialog	A dialog to query user's preferences for new entity classes.
AddEntitiesOrManageElementsDialog	A dialog to query user's preferences for new entities.
AddEntitiesDialog	A dialog to query user's preferences for new entities.
ManageElementsDialog	A dialog to query user's preferences for managing entity dimensions.
EntityGroupDialogBase	**param parent** data store widget
AddEntityGroupDialog	**param parent** data store widget
ManageMembersDialog	**param parent** data store widget

class spinetoolbox.spine_db_editor.widgets.add_items_dialogs.**AddReadyEntitiesDialog**(*parent*, *entity_class*, *entities*, *db_mngr*, **db_maps*, *commit_data=True*)

Bases: *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.DialogWithTableAndButtons*

A dialog to let the user add new 'ready' multidimensional entities.

Parameters

- **parent** (SpineDBEditor) –
- **entity_class** (*dict*) –
- **entities** (*list(list(str))*) –
- **db_mngr** (SpineDBManager) –
- ***db_maps** – DatabaseMapping instances

_populate_layout()

make_table_view()

populate_table_view()

connect_signals()

Connect signals to slots.

_handle_table_view_cell_clicked(*row*, *column*)

_handle_table_view_current_changed(*current*, *_previous*)

accept()

Collect info from dialog and try to add items.

get_db_map_data()

class spinetoolbox.spine_db_editor.widgets.add_items_dialogs.**AddItemsDialog**(*parent*, *db_mngr*, **db_maps*)

Bases: *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.ManageItemsDialog*

A dialog to query user's preferences for new db items.

Parameters

- **parent** (SpineDBEditor) –
- **db_mngr** (SpineDBManager) –
- ***db_maps** – DiffDatabaseMapping instances

_populate_layout()

connect_signals()

Connect signals to slots.

remove_selected_rows(*checked=True*)

all_databases(*row*)

Returns a list of db names available for a given row. Used by delegates.

class spinetoolbox.spine_db_editor.widgets.add_items_dialogs.**AddEntityClassesDialog**(*parent*, *item*, *db_mngr*, **db_maps*, *force_default=False*)

Bases: *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.ShowIconColorEditorMixin*, *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.GetEntityClassesMixin*, *AddItemsDialog*

A dialog to query user's preferences for new entity classes.

Parameters

- **parent** (SpineDBEditor) –
- **item** (MultiDBTreeItem) –
- **db_mngr** (SpineDBManager) –
- ***db_maps** – DatabaseMapping instances
- **force_default** (*bool*) – if True, defaults are non-editable

_populate_layout()

connect_signals()

Connect signals to slots.

_handle_spin_box_value_changed(*i*)

insert_column()

remove_column()

_handle_model_data_changed(*top_left*, *bottom_right*, *roles*)

Reimplement in subclasses to handle changes in model data.

accept()

Collect info from dialog and try to add items.

class spinetoolbox.spine_db_editor.widgets.add_items_dialogs.**AddEntitiesOrManageElementsDialog**(*parent*, *db_mngr*, **db_maps*)

Bases: *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.GetEntityClassesMixin*, *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.GetEntitiesMixin*, *AddItemsDialog*

A dialog to query user's preferences for new entities.

Parameters

- **parent** (SpineDBEditor) –
- **db_mngr** (SpineDBManager) –
- ***db_maps** – DatabaseMapping instances

abstract _class_key_to_str(*key*, **db_maps*)

abstract _accepts_class(*ent_cls*)

Returns whether the widget should handle the given entity class.

Parameters
ent_cls (*MappedItem*) –

Returns
bool

abstract make_model()

abstract _do_reset_model()

reset_model(*index*)

Setup model according to current entity class selected in combobox.

_populate_layout()

connect_signals()

Connect signals to slots.

_handle_model_data_changed(*top_left*, *bottom_right*, *roles*)

Reimplement in subclasses to handle changes in model data.

class spinetoolbox.spine_db_editor.widgets.add_items_dialogs.**AddEntitiesDialog**(*parent*, *item*, *db_mngr*, **db_maps*, *force_default=False*, *commit_data=True*)

Bases: *AddEntitiesOrManageElementsDialog*

A dialog to query user's preferences for new entities.

Parameters

- **parent** (SpineDBEditor) –
- **item** (MultiDBTreeItem) –
- **db_mngr** (SpineDBManager) –
- ***db_maps** – DatabaseMapping instances
- **force_default** (*bool*) – if True, defaults are non-editable

_class_key_to_str(*key*, **db_maps*)

_accepts_class(*ent_cls*)

Returns whether the widget should handle the given entity class.

Parameters
ent_cls (*MappedItem*) –

Returns
bool

make_model()

_do_reset_model()

get_db_map_data()

accept()

Collect info from dialog and try to add items.

class spinetoolbox.spine_db_editor.widgets.add_items_dialogs.**ManageElementsDialog**(*parent*, *item*, *db_mngr*, **db_maps*)

Bases: *AddEntitiesOrManageElementsDialog*

A dialog to query user's preferences for managing entity dimensions.

Parameters

- **parent** (SpineDBEditor) – data store widget
- **item** (MultiDBTreeItem) –
- **db_mngr** (SpineDBManager) – the manager to do the removal
- ***db_maps** – DatabaseMapping instances

_populate_layout()

make_model()

splitter_widgets()

connect_signals()

Connect signals to slots.

_accepts_class(*ent_cls*)

Returns whether the widget should handle the given entity class.

Parameters
ent_cls (*MappedItem*) –

Returns
bool

_class_key_to_str(*key*, **_db_maps*)

reset_entity_class_combo_box(*database*)

add_entities(*checked=True*)

_do_reset_model()

resize_window_to_columns(*height=None*)

accept()

Collect info from dialog and try to add items.

class spinetoolbox.spine_db_editor.widgets.add_items_dialogs.**EntityGroupDialogBase**(*parent*, *entity_class_item*, *db_mngr*, **db_maps*)

Bases: PySide6.QtWidgets.QDialog

Parameters

- **parent** (SpineDBEditor) – data store widget
- **entity_class_item** (EntityClassItem) –
- **db_mngr** (SpineDBManager) –
- ***db_maps** – database mappings

connect_signals()

Connect signals to slots.

reset_list_widgets(*database*)

abstract initial_member_ids()

abstract initial_entity_id()

add_members(*checked=False*)

remove_members(*checked=False*)

_check_validity()

class spinetoolbox.spine_db_editor.widgets.add_items_dialogs.**AddEntityGroupDialog**(*parent*, *entity_class_item*, *db_mngr*, **db_maps*)

Bases: *EntityGroupDialogBase*

Parameters

- **parent** (SpineDBEditor) – data store widget
- **entity_class_item** (EntityClassItem) –
- **db_mngr** (SpineDBManager) –
- ***db_maps** – database mappings

initial_member_ids()

initial_entity_id()

_check_validity()

accept()

class spinetoolbox.spine_db_editor.widgets.add_items_dialogs.**ManageMembersDialog**(*parent*, *entity_item*, *db_mngr*, **db_maps*)

Bases: *EntityGroupDialogBase*

Parameters

- **parent** (SpineDBEditor) – data store widget
- **entity_item** (entity_tree_item.EntityItem) –
- **db_mngr** (SpineDBManager) –
- ***db_maps** – database mappings

_entity_groups()

initial_member_ids()

initial_entity_id()

accept()

spinetoolbox.spine_db_editor.widgets.commit_viewer

Contains Database editor's Commit viewer.

Module Contents

Classes

_DBCommitViewer	Commit viewer's central widget.
_CommitItem	A widget to show commit message, author and data on a QTreeWidget.
_AffectedItemsWidget	A composite widget that contains a table and a label.
CommitViewer	Commit viewer window.
Worker	Worker that fetches affected items.

class spinetoolbox.spine_db_editor.widgets.commit_viewer.**_DBCommitViewer**(*db_mngr*, *db_map*, *parent=None*)

Bases: PySide6.QtWidgets.QWidget

Commit viewer's central widget.

Parameters

- **db_mngr** (SpineDBManager) – database manager
- **db_map** (*DatabaseMapping*) – database mapping
- **parent** (*QWidget, optional*) – parent widget

property splitter: PySide6.QtWidgets.QSplitter

_select_commit(*current*, *previous*)

Start a worker thread that fetches affected items for the selected commit.

Parameters

- **current** (*QTreeWidgetItem*) – currently selected commit item
- **previous** (*QTreeWidgetItem*) – previously selected commit item

_launch_new_worker(*commit_id*)

Starts a new worker thread.

If a thread is already running, it is quite before starting a new one.

Parameters
commit_id (*TempId*) – commit id

_process_affected_items(*item_type*, *keys*, *items*)

Adds a fetched chunk of affected items to appropriate table view.

Parameters

- **item_type** (*str*) – fethced item type
- **keys** (*Sequence of str*) – item keys
- **items** (*Sequence of Sequence*) – list of items, each item being a list of labels; items must have the same length as keys

_max_affected_items_fetched(*item_type*, *still_available*)

Updates the fetch status label.

Parameters

- **item_type** (*str*) – item type
- **still_available** (*int*) – number of items left unfetched

_all_affected_items_fetched(*item_type*)

Hides the fetch status label.

Parameters
item_type (*str*) – item type

_finish_work()

Quits the worker thread if it is running.

tear_down()

Tears down the widget.

_set_preferred_item_type(*preferred_tab_index*)

Sets the preferred item type for affected items.

The tab showing the preferred type is selected automatically as the current tab when/if it gets fetched.

Parameters
preferred_tab_index (*int*) – index of the preferred tab

class spinetoolbox.spine_db_editor.widgets.commit_viewer._CommitItem(*commit*, *parent=None*)

Bases: PySide6.QtWidgets.QWidget

A widget to show commit message, author and data on a QTreeWidget.

Parameters

- **commit** (*dict*) – commit database item
- **parent** (*QWidget, optional*) – parent widget

class spinetoolbox.spine_db_editor.widgets.commit_viewer._AffectedItemsWidget

Bases: PySide6.QtWidgets.QWidget

A composite widget that contains a table and a label.

property table: PySide6.QtWidgets.QTableWidget

property label: PySide6.QtWidgets.QLabel

class spinetoolbox.spine_db_editor.widgets.commit_viewer.CommitViewer(*qsettings*, *db_mngr*, **db_maps*, *parent=None*)

Bases: PySide6.QtWidgets.QMainWindow

Commit viewer window.

Parameters

- **qsettings** (*QSettings*) – application settings
- **db_mngr** (SpineDBManager) – database manager
- ***db_maps** – database mappings to view
- **parent** (*QWidget, optional*) – parent widget

_carry_splitter_state(*index*)

Ensures that splitters have the same state in all tabs.

Parameters
index (*int*) – current database tab index

closeEvent(*ev*)

class spinetoolbox.spine_db_editor.widgets.commit_viewer.**Worker**(*db_mngr*, *db_map*, *commit_id*)

Bases: PySide6.QtCore.QObject

Worker that fetches affected items.

The items are fetched in chunks which makes it possible to quit the thread mid-execution. There is also a hard limit to how many items are fetched.

Parameters

- **db_mngr** (SpineDBManager) – database manager
- **db_map** (*DatabaseMapping*) – database mapping
- **commit_id** (*TempId*) – commit id

SOFT_MAX_IDS = 400

HARD_EXTRA_ID_LIMIT = 100

CHUNK_SIZE = 50

max_ids_reached

all_ids_fetched

chunk_ready

finished

run()

Fetches affected items.

static _parse_value(*db_mngr*, *db_map*, *item*, *key*)

Converts item field values to something more displayable.

Parameters

- **db_mngr** (SpineDBManager) – database manager
- **db_map** (*DatabaseMapping*) – database mapping
- **item** (*PublicItem*) – database item
- **key** (*str*) – value's key

Returns
displayable presentation of the value

Return type
str

spinetoolbox.spine_db_editor.widgets.custom_delegates

Custom item delegates.

Module Contents

Classes

PivotTableDelegateMixin	A mixin that fixes Pivot table's header table editor position.
RelationshipPivotTableDelegate	A mixin that fixes Pivot table's header table editor position.
ScenarioAlternativeTableDelegate	A mixin that fixes Pivot table's header table editor position.
ParameterPivotTableDelegate	A mixin that fixes Pivot table's header table editor position.
ParameterValueElementDelegate	Delegate for Array and Map editors' table cells.
TableDelegate	Base class for all custom stacked table delegates.
DatabaseNameDelegate	A delegate for the database name.
ParameterValueOrDefaultValueDelegate	A delegate for either the value or the default value.
ParameterDefaultValueDelegate	A delegate for the default value.
ParameterValueDelegate	A delegate for the parameter_value.
ValueListDelegate	A delegate for the parameter value list.
EntityClassNameDelegate	A delegate for the object_class name.
ParameterNameDelegate	A delegate for the object parameter name.
EntityBynameDelegate	A delegate for the entity byname.
AlternativeNameDelegate	A delegate for the alternative name.
BooleanValueDelegate	Base class for all custom stacked table delegates.
AlternativeDelegate	A delegate for the alternative tree.
ScenarioDelegate	A delegate for the scenario tree.
ParameterDefinitionNameAndDescriptionDelegate	A delegate for the parameter_name and description columns in Parameter Definition Table View.
ParameterValueListDelegate	A delegate for the parameter value list tree.
ManageItemsDelegate	A custom delegate for the model in {Add/Edit}ItemDialogs.
ManageEntityClassesDelegate	A delegate for the model and view in {Add/Edit}EntityClassesDialog.
ManageEntitiesDelegate	A delegate for the model and view in {Add/Edit}EntitiesDialog.
RemoveEntitiesDelegate	A delegate for the model and view in RemoveEntities-Dialog.
MetadataDelegate	A delegate for the name and value columns in Metadata Table View.
ItemMetadataDelegate	A delegate for name and value columns in item metadata editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**PivotTableDelegateMixin**

A mixin that fixes Pivot table's header table editor position.

updateEditorGeometry(*editor*, *option*, *index*)

Fixes position of header table editors.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**RelationshipPivotTableDelegate**(*parent*)

Bases: *PivotTableDelegateMixin*, *spinetoolbox.widgets.custom_delegates.CheckBoxDelegate*

A mixin that fixes Pivot table's header table editor position.

Parameters
parent (SpineDBEditor) – parent widget, i.e. the database editor

data_committed

static _is_relationship_index(*index*)

Checks whether the given index corresponds to a relationship, in which case we need to use the check box delegate.

Parameters
index (*QModelIndex*) – index to check

Returns
True if index corresponds to relationship, False otherwise

Return type
bool

setModelData(*editor*, *model*, *index*)

Send signal.

setEditorData(*editor*, *index*)

Do nothing. We're setting editor data right away in createEditor.

paint(*painter*, *option*, *index*)

Paint a checkbox without the label.

editorEvent(*event*, *model*, *option*, *index*)

Change the data in the model and the state of the checkbox when user presses left mouse button and this cell is editable. Otherwise do nothing.

createEditor(*parent*, *option*, *index*)

Important, otherwise an editor is created if the user clicks in this cell. ** Need to hook up a signal to the model.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ScenarioAlternativeTableDelegate**(*parent*)

Bases: *PivotTableDelegateMixin*, *spinetoolbox.widgets.custom_delegates.RankDelegate*

A mixin that fixes Pivot table's header table editor position.

Parameters
parent (SpineDBEditor) – database editor

data_committed

static _is_scenario_alternative_index(*index*)

Checks whether or not the given index corresponds to a scenario alternative, in which case we need to use the rank delegate.

Returns
bool

setModelData(*editor*, *model*, *index*)

Send signal.

setEditorData(*editor*, *index*)

Do nothing. We're setting editor data right away in createEditor.

paint(*painter*, *option*, *index*)

Paint a checkbox without the label.

editorEvent(*event*, *model*, *option*, *index*)

Change the data in the model and the state of the checkbox when user presses left mouse button and this cell is editable. Otherwise do nothing.

createEditor(*parent*, *option*, *index*)

Important, otherwise an editor is created if the user clicks in this cell. ** Need to hook up a signal to the model.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ParameterPivotTableDelegate**(*parent*)

Bases: *PivotTableDelegateMixin*, PySide6.QtWidgets.QStyledItemDelegate

A mixin that fixes Pivot table's header table editor position.

Parameters
parent (SpineDBEditor) – parent widget, i.e. database editor

parameter_value_editor_requested

data_committed

setModelData(*editor*, *model*, *index*)

Send signal.

setEditorData(*editor*, *index*)

Do nothing. We're setting editor data right away in createEditor.

createEditor(*parent*, *option*, *index*)

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ParameterValueElementDelegate**

Bases: PySide6.QtWidgets.QStyledItemDelegate

Delegate for Array and Map editors' table cells.

value_editor_requested

Emitted when editing the value requires the full blown editor dialog.

setModelData(*editor*, *model*, *index*)

Sets data in the model.

editor (CustomLineEditor): editor widget model (QAbstractItemModel): model index (QModelIndex): target index

createEditor(*parent*, *option*, *index*)

Creates an editor widget or emits value_editor_requested for complex values.

Parameters

- **parent** (*QWidget*) – parent widget
- **option** (*QStyleOptionViewItem*) – unused
- **index** (*QModelIndex*) – element's model index

Returns
editor widget

Return type
ParameterValueLineEditor

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**TableDelegate**(*parent*, *db_mngr*)

Bases: PySide6.QtWidgets.QStyledItemDelegate

Base class for all custom stacked table delegates.

db_mngr

database manager

Type
SpineDBManager

Parameters

- **parent** (*QWidget*) – parent widget
- **db_mngr** (SpineDBManager) – database manager

data_committed

setModelData(*editor*, *model*, *index*)

Send signal.

setEditorData(*editor*, *index*)

Do nothing. We're setting editor data right away in createEditor.

updateEditorGeometry(*editor*, *option*, *index*)

_close_editor(*editor*, *index*)

Closes editor. Needed by SearchBarEditor.

_get_db_map(*index*)

Returns the db_map for the database at given index or None if not set yet.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**DatabaseNameDelegate**(*parent*, *db_mngr*)

Bases: *TableDelegate*

A delegate for the database name.

Parameters

- **parent** (*QWidget*) – parent widget
- **db_mngr** (SpineDBManager) – database manager

createEditor(*parent*, *option*, *index*)

Returns editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ParameterValueOrDefaultValueDelegate**(*parent*, *db_mngr*)

Bases: *TableDelegate*

A delegate for either the value or the default value.

Parameters

- **parent** (*QWidget*) – parent widget
- **db_mngr** (*SpineDatabaseManager*) – database manager

parameter_value_editor_requested

setModelData(*editor*, *model*, *index*)

Send signal.

_create_or_request_parameter_value_editor(*parent*, *index*)

Emits the signal to request a standalone *ParameterValueEditor* from parent widget.

Parameters

- **parent** (*QWidget*) – editor's parent widget
- **index** (*QModelIndex*) – index to parameter value model

Returns
editor or None if `parameter_value_editor_request` signal was emitted

Return type
ParameterValueLineEditor

abstract _get_value_list_id(*index*, *db_map*)

Returns a value list id for the given index and db_map.

Parameters

- **index** (*QModelIndex*) – value list's index
- **db_map** (*DiffDatabaseMapping*) – database mapping

Returns
value list id

Return type
int

createEditor(*parent*, *option*, *index*)

If the parameter has associated a value list, returns a SearchBarEditor. Otherwise, returns or requests a dedicated parameter_value editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ParameterDefaultValueDelegate**(*parent*, *db_mngr*)

Bases: *ParameterValueOrDefaultValueDelegate*

A delegate for the default value.

Parameters

- **parent** (*QWidget*) – parent widget
- **db_mngr** (*SpineDatabaseManager*) – database manager

_get_value_list_id(*index*, *db_map*)

See base class

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ParameterValueDelegate**(*parent*, *db_mngr*)

Bases: *ParameterValueOrDefaultValueDelegate*

A delegate for the parameter_value.

Parameters

- **parent** (*QWidget*) – parent widget

- **db_mngr** (*SpineDatabaseManager*) – database manager

_get_value_list_id(*index*, *db_map*)

See base class.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ValueListDelegate**(*parent*, *db_mngr*)

Bases: *TableDelegate*

A delegate for the parameter value list.

Parameters

- **parent** (*QWidget*) – parent widget
- **db_mngr** (SpineDBManager) – database manager

createEditor(*parent*, *option*, *index*)

Returns editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**EntityClassNameDelegate**(*parent*, *db_mngr*)

Bases: *TableDelegate*

A delegate for the object_class name.

Parameters

- **parent** (*QWidget*) – parent widget
- **db_mngr** (SpineDBManager) – database manager

createEditor(*parent*, *option*, *index*)

Returns editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ParameterNameDelegate**(*parent*, *db_mngr*)

Bases: *TableDelegate*

A delegate for the object parameter name.

Parameters

- **parent** (*QWidget*) – parent widget
- **db_mngr** (SpineDBManager) – database manager

createEditor(*parent*, *option*, *index*)

Returns editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**EntityBynameDelegate**(*parent*, *db_mngr*)

Bases: *TableDelegate*

A delegate for the entity byname.

Parameters

- **parent** (*QWidget*) – parent widget
- **db_mngr** (SpineDBManager) – database manager

element_name_list_editor_requested

createEditor(*parent*, *option*, *index*)

Returns editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**AlternativeNameDelegate**(*parent*, *db_mngr*)

Bases: *TableDelegate*

A delegate for the alternative name.

Parameters

- **parent** (*QWidget*) – parent widget
- **db_mngr** (SpineDBManager) – database manager

createEditor(*parent*, *option*, *index*)

Returns editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**BooleanValueDelegate**(*parent*, *db_mngr*)

Bases: *TableDelegate*

Base class for all custom stacked table delegates.

db_mngr

database manager

Type
SpineDBManager

Parameters

- **parent** (*QWidget*) – parent widget
- **db_mngr** (SpineDBManager) – database manager

setModelData(*editor*, *model*, *index*)

Sends signal.

createEditor(*parent*, *option*, *index*)

Returns editor.

classmethod make_editor(*parent*, *tutor*, *index*)

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**AlternativeDelegate**

Bases: PySide6.QtWidgets.QStyledItemDelegate

A delegate for the alternative tree.

data_committed

setModelData(*editor*, *model*, *index*)

Send signal.

setEditorData(*editor*, *index*)

Do nothing. We're setting editor data right away in createEditor.

createEditor(*parent*, *option*, *index*)

Returns editor.

_close_editor(*editor*, *index*)

Closes editor.

Needed by SearchBarEditor.

Parameters

- **editor** (*QWidget*) – editor widget
- **index** (*QModelIndex*) – index that is being edited

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ScenarioDelegate**(**args*, ***kwargs*)

Bases: PySide6.QtWidgets.QStyledItemDelegate

A delegate for the scenario tree.

Parameters

- ***args** – arguments passed to QStyledItemDelegate
- ****kwargs** – keyword arguments passed to QStyledItemDelegate

data_committed

setModelData(*editor*, *model*, *index*)

Send signal.

setEditorData(*editor*, *index*)

Do nothing. We're setting editor data right away in createEditor.

_update_alternative_ids(*item*)

Updates available alternatives avoiding duplicates in a scenario.

Excludes alternatives that are already in the scenario

Parameters
item (ScenarioAlternativeItem) – one of scenario's scenario alternatives

Returns
available alternative names

Return type
list of str

createEditor(*parent*, *option*, *index*)

Returns editor.

updateEditorGeometry(*editor*, *option*, *index*)

_close_editor(*editor*, *index*)

Closes editor.

Needed by SearchBarEditor.

Parameters

- **editor** (*QWidget*) – editor widget
- **index** (*QModelIndex*) – index that is being edited

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ParameterDefinitionNameAndDescriptionDelega**

Bases: *TableDelegate*

A delegate for the parameter_name and description columns in Parameter Definition Table View.

Parameters

- **parent** (*QWidget*) – parent widget
- **db_mngr** (SpineDBManager) – database manager

setEditorData(*editor*, *index*)

Do nothing. We're setting editor data right away in createEditor.

createEditor(*parent*, *option*, *index*)

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ParameterValueListDelegate**

Bases: PySide6.QtWidgets.QStyledItemDelegate

A delegate for the parameter value list tree.

data_committed

parameter_value_editor_requested

setModelData(*editor*, *model*, *index*)

Send signal.

setEditorData(*editor*, *index*)

Do nothing. We're setting editor data right away in createEditor.

createEditor(*parent*, *option*, *index*)

Returns editor.

_close_editor(*editor*, *index*)

Closes editor.

Needed by SearchBarEditor.

Parameters

- **editor** (*QWidget*) – editor widget
- **index** (*QModelIndex*) – index that is being edited

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ManageItemsDelegate**

Bases: PySide6.QtWidgets.QStyledItemDelegate

A custom delegate for the model in {Add/Edit}ItemDialogs.

data_committed

setModelData(*editor*, *model*, *index*)

Send signal.

close_editor(*editor*, *index*)

Closes editor.

Needed by SearchBarEditor.

Parameters

- **editor** (*QWidget*) – editor widget

- **index** (*QModelIndex*) – index that is being edited

updateEditorGeometry(*editor*, *option*, *index*)

connect_editor_signals(*editor*, *index*)

Connect editor signals if necessary.

Parameters

- **editor** (*QWidget*) – editor widget
- **index** (*QModelIndex*) – index being edited

_create_database_editor(*parent*, *index*)

Creates an editor.

Parameters

- **parent** (*QWidget*) – parent widget
- **index** (*QModelIndex*) – index being edited

Returns
editor

Return type
QWidget

createEditor(*parent*, *option*, *index*)

Returns an editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ManageEntityClassesDelegate**

Bases: *ManageItemsDelegate*

A delegate for the model and view in {Add/Edit}EntityClassesDialog.

icon_color_editor_requested

paint(*painter*, *option*, *index*)

Get a pixmap from the index data and paint it in the middle of the cell.

createEditor(*parent*, *option*, *index*)

Return editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ManageEntitiesDelegate**

Bases: *ManageItemsDelegate*

A delegate for the model and view in {Add/Edit}EntitiesDialog.

createEditor(*parent*, *option*, *index*)

Return editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**RemoveEntitiesDelegate**

Bases: *ManageItemsDelegate*

A delegate for the model and view in RemoveEntitiesDialog.

createEditor(*parent*, *option*, *index*)

Return editor.

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**MetadataDelegate**

Bases: PySide6.QtWidgets.QStyledItemDelegate

A delegate for the name and value columns in Metadata Table View.

setEditorData(*editor*, *index*)

createEditor(*parent*, *option*, *index*)

class spinetoolbox.spine_db_editor.widgets.custom_delegates.**ItemMetadataDelegate**(*item_metadata_model*, *metadata_model*, *column*, *parent*)

Bases: PySide6.QtWidgets.QStyledItemDelegate

A delegate for name and value columns in item metadata editor.

Parameters

- **item_metadata_model** (*ItemMetadataModel*) – item metadata model
- **metadata_model** (MetadataTableModel) – metadata model
- **column** (*int*) – item metadata table column
- **parent** (*QObject, optional*) – parent object

createEditor(*parent*, *option*, *index*)

spinetoolbox.spine_db_editor.widgets.custom_editors

Custom editors for model/view programming.

Module Contents

Classes

EventFilterForCatchingRollbackShortcut	
CustomComboBoxEditor	
CustomLineEditor	A custom QLineEdit to handle data from models.
ParameterValueLineEditor	A custom QLineEdit to handle data from models.
PivotHeaderTableLineEditor	Line editor that is visible on Pivot view's header tables due to a clever hack.
_CustomLineEditDelegate	A delegate for placing a CustomLineEditor on the first row of SearchBarEditor.
SearchBarEditor	A Google-like search bar, implemented as a QTableView with a _CustomLineEditDelegate in the first row.
BooleanSearchBarEditor	A Google-like search bar, implemented as a QTableView with a _CustomLineEditDelegate in the first row.
CheckListEditor	A check list editor.
_IconPainterDelegate	A delegate to highlight decorations in a QListWidget.
IconColorEditor	An editor to let the user select an icon and a color for an object_class.

class spinetoolbox.spine_db_editor.widgets.custom_editors.**EventFilterForCatchingRollbackShortcut**

Bases: PySide6.QtCore.QObject

eventFilter(*obj*, *event*)

Catches Rollback action shortcut (Ctrl+backspace) while editing is in progress.

class spinetoolbox.spine_db_editor.widgets.custom_editors.**CustomComboBoxEditor**(*parent*)

Bases: PySide6.QtWidgets.QComboBox

class spinetoolbox.spine_db_editor.widgets.custom_editors.**CustomLineEditor**(*parent*)

Bases: PySide6.QtWidgets.QLineEdit

A custom QLineEdit to handle data from models.

set_data(*data*)

Sets editor's text.

Parameters
data (*Any*) – anything convertible to string

data()

Returns editor's text.

Returns
editor's text

Return type
str

keyPressEvent(*event*)

Prevents shift key press to clear the contents.

class spinetoolbox.spine_db_editor.widgets.custom_editors.**ParameterValueLineEditor**(*parent*)

Bases: *CustomLineEditor*

A custom QLineEdit to handle data from models.

set_data(*data*)

See base class.

data()

See base class.

class spinetoolbox.spine_db_editor.widgets.custom_editors.**PivotHeaderTableLineEditor**(*parent=None*)

Bases: *CustomLineEditor*

Line editor that is visible on Pivot view's header tables due to a clever hack.

Parameters
parent (*QWidget, optional*) – parent widget

fix_geometry()

Fixes editor's position after reparenting.

class spinetoolbox.spine_db_editor.widgets.custom_editors.**_CustomLineEditDelegate**

Bases: PySide6.QtWidgets.QStyledItemDelegate

A delegate for placing a CustomLineEditor on the first row of SearchBarEditor.

text_edited

setModelData(*editor*, *model*, *index*)

createEditor(*parent*, *option*, *index*)

Create editor and 'forward' *textEdited* signal.

eventFilter(*editor*, *event*)

Handle all sort of special cases.

class spinetoolbox.spine_db_editor.widgets.custom_editors.**SearchBarEditor**(*parent*, *tutor=None*)

Bases: PySide6.QtWidgets.QTableView

A Google-like search bar, implemented as a QTableView with a _CustomLineEditDelegate in the first row.

Parameters

- **parent** (*QWidget, optional*) – parent widget
- **tutor** (*QWidget, optional*) – another widget used for positioning.

data_committed

set_data(*current*, *items*)

Populates model.

Parameters

- **current** (*str*) – item that is currently selected from given items
- **items** (*Sequence of str*) – items to show in the list

set_base_offset(*offset*)

Changes the base offset that is applied to the editor's position.

Parameters
offset (*QPoint*) – new offset

update_geometry(*option*)

Updates geometry.

Parameters
option (*QStyleOptionViewItem*) – style information

refit()

Changes the position and size of the editor to fit the window.

data()

Returns editor's final data.

Returns
editor data

Return type
str

_handle_delegate_text_edited(*text*)

Filters model as the first row is being edited.

Parameters
text (*str*) – text the user has entered on the first row

_proxy_model_filter_accepts_row(*source_row*, *source_parent*)

Always accept first row while filtering the rest.

Parameters

- **source_row** (*int*) – source row index
- **source_parent** (*QModelIndex*) – parent index for source row

Returns
True if row is accepted, False otherwise

Return type
bool

keyPressEvent(*event*)

Sets data from current index into first index as the user navigates through the table using the up and down keys.

currentChanged(*current*, *previous*)

edit_first_index()

Edits first index if valid and not already being edited.

mousePressEvent(*event*)

Commits data.

class spinetoolbox.spine_db_editor.widgets.custom_editors.**BooleanSearchBarEditor**(*parent*, *tutor=None*)

Bases: *SearchBarEditor*

A Google-like search bar, implemented as a QTableView with a _CustomLineEditDelegate in the first row.

Parameters

- **parent** (*QWidget, optional*) – parent widget
- **tutor** (*QWidget, optional*) – another widget used for positioning.

data()

Returns editor's final data.

Returns
editor data

Return type
str

set_data(*current*, *items*)

Populates model.

Parameters

- **current** (*str*) – item that is currently selected from given items
- **items** (*Sequence of str*) – items to show in the list

class spinetoolbox.spine_db_editor.widgets.custom_editors.**CheckListEditor**(*parent*, *tutor=None*)

Bases: PySide6.QtWidgets.QTableView

A check list editor.

Parameters

- **parent** (*QWidget*) – parent widget
- **tutor** (*QWidget, optional*) – a widget that helps in positioning

keyPressEvent(*event*)

Toggles checked state if the user presses space.

toggle_selected(*index*)

Adds or removes given index from selected items.

Parameters
index (*QModelIndex*) – index to toggle

mouseMoveEvent(*event*)

Sets the current index to the one under mouse.

mousePressEvent(*event*)

Toggles checked state of pressed index.

set_data(*items*, *checked_items*)

Sets data and updates geometry.

Parameters

- **items** (*Sequence(str)*) – All items.
- **checked_items** (*Sequence(str)*) – Initially checked items.

data()

Returns a comma separated list of checked items.

Returns
str

update_geometry(*option*)

Updates geometry.

Parameters
option (*QStyleOptionViewItem*) – style information

class spinetoolbox.spine_db_editor.widgets.custom_editors.**_IconPainterDelegate**

Bases: PySide6.QtWidgets.QStyledItemDelegate

A delegate to highlight decorations in a QListWidget.

paint(*painter*, *option*, *index*)

Paints selected items using the highlight brush.

class spinetoolbox.spine_db_editor.widgets.custom_editors.**IconColorEditor**(*parent*)

Bases: PySide6.QtWidgets.QDialog

An editor to let the user select an icon and a color for an object_class.

Parameters
parent (*QWidget*) – parent widget

_proxy_model_filter_accepts_row(*source_row*, *source_parent*)

Filters icons according to search terms.

Parameters

- **source_row** (*int*) – source row index
- **source_parent** (*QModelIndex*) – parent index for source row

Returns
True if row is accepted, False otherwise

Return type
bool

connect_signals()

Connects signals to slots.

set_data(*data*)

Sets current icon data.

Parameters
data (*int*) – database icon data

data()

Gets current icon data.

Returns
database icon data

Return type
int

spinetoolbox.spine_db_editor.widgets.custom_menus

Classes for custom context menus and pop-up menus.

Module Contents

Classes

MainMenu	

AutoFilterMenu	Filter menu.
TabularViewFilterMenuBase	Filter menu.
TabularViewDBItemFilterMenu	Filter menu to use together with FilterWidget in TabularViewMixin.
TabularViewCodenameFilterMenu	Filter menu to filter database codenames in Pivot table.

class spinetoolbox.spine_db_editor.widgets.custom_menus.**MainMenu**

Bases: PySide6.QtWidgets.QMenu

event(*ev*)

Intercepts shortcuts and instead sends an equivalent event with the 'Alt' modifier, so that mnemonics works with just the key. Also sends a key press event with the 'Alt' key when this menu shows, so that mnemonics are underlined on Windows.

class spinetoolbox.spine_db_editor.widgets.custom_menus.**AutoFilterMenu**(*parent*, *db_mngr*, *db_maps*, *item_type*, *field*, *show_empty=True*)

Bases: *spinetoolbox.widgets.custom_menus.FilterMenuBase*

Filter menu.

Parameters

- **parent** (SpineDBEditor) –
- **db_mngr** (SpineDBManager) –
- **db_maps** (*Sequence of DatabaseMapping*) –
- **item_type** (*str*) –
- **field** (*str*) – the field name

filterChanged

set_filter_accepted_values(*accepted_values*)

set_filter_rejected_values(*rejected_values*)

_get_value(*item*, *db_map*)

_get_display_value(*item*, *db_map*)

_handle_items_added(*db_map_data*)

_handle_items_removed(*db_map_data*)

_build_auto_filter(*valid_values*)

Builds the auto filter given valid values.

Parameters
valid_values (*Sequence*) – Values accepted by the filter.

Returns
mapping (db_map, entity_class_id) to set of valid values

Return type
dict

emit_filter_changed(*valid_values*)

Builds auto filter and emits signal.

Parameters
valid_values (*Sequence*) – Values accepted by the filter.

class spinetoolbox.spine_db_editor.widgets.custom_menus.**TabularViewFilterMenuBase**(*parent*, *identifier*)

Bases: *spinetoolbox.widgets.custom_menus.FilterMenuBase*

Filter menu.

Parameters

- **parent** (SpineDBEditor) – parent widget
- **identifier** (*str*) – header identifier

filterChanged

showEvent(*event*)

class spinetoolbox.spine_db_editor.widgets.custom_menus.**TabularViewDBItemFilterMenu**(*parent*, *db_mngr*, *db_maps*, *item_type*, *accepts_item*, *identifier*, *show_empty=True*)

Bases: *TabularViewFilterMenuBase*

Filter menu to use together with FilterWidget in TabularViewMixin.

Parameters

- **parent** (SpineDBEditor) – parent widget
- **db_mngr** (SpineDBManager) – database manager
- **db_maps** (*Sequence of DatabaseMapping*) – database mappings
- **item_type** (*str*) – database item type to filter
- **accepts_item** (*Callable*) – callable that returns True when database item is accepted
- **identifier** (*str*) – header identifier

- **show_empty** (*bool*) – if True, an empty row will be added to the end of the item list

_handle_items_added(*db_map_data*)

_get_values(*db_map*, *item*)

_handle_items_removed(*db_map_data*)

emit_filter_changed(*valid_values*)

class spinetoolbox.spine_db_editor.widgets.custom_menus.**TabularViewCodenameFilterMenu**(*parent*, *db_maps*, *identifier*, *show_empty=True*)

Bases: *TabularViewFilterMenuBase*

Filter menu to filter database codenames in Pivot table.

Parameters

- **parent** (SpineDBEditor) – parent widget
- **db_maps** (*Sequence of DatabaseMapping*) – database mappings
- **identifier** (*str*) – header identifier
- **show_empty** (*bool*) – if True, an empty row will be added to the end of the item list

emit_filter_changed(*valid_values*)

See base class.

spinetoolbox.spine_db_editor.widgets.custom_qgraphicsviews

Classes for custom QGraphicsViews for the Entity graph view.

Module Contents

Classes

_GraphProperty	
_GraphBoolProperty	
_GraphIntProperty	
EntityQGraphicsView	QGraphicsView for the Entity Graph View.

class spinetoolbox.spine_db_editor.widgets.custom_qgraphicsviews.**_GraphProperty**(*name*, *settings_name*)

property value

connect_spine_db_editor(*spine_db_editor*)

class spinetoolbox.spine_db_editor.widgets.custom_qgraphicsviews._GraphBoolProperty(*args, **kwargs)

Bases: _GraphProperty

_set_value(_checked=False, save_setting=True)

set_value(checked)

update(menu)

class spinetoolbox.spine_db_editor.widgets.custom_qgraphicsviews._GraphIntProperty(min_value, max_value, default_value, *args, **kwargs)

Bases: _GraphProperty

_set_value(_value=None, save_setting=True)

set_value(value)

update(menu)

class spinetoolbox.spine_db_editor.widgets.custom_qgraphicsviews.EntityQGraphicsView(parent)

Bases: spinetoolbox.widgets.custom_qgraphicsviews.CustomQGraphicsView

QGraphicsView for the Entity Graph View.

Parameters
parent (*QWidget*) – Graph View Form's (QMainWindow) central widget (self.centralwidget)

property _qsettings

property db_mngr

property entity_items

graph_selection_changed

get_property(name)

set_property(name, value)

get_all_properties()

set_many_properties(props)

handle_scene_selection_changed()

Filters parameters by selected objects in the graph.

connect_spine_db_editor(spine_db_editor)

populate_context_menu()

_update_actions_visibility()

Enables or disables actions according to current selection in the graph.

make_items_menu()

_save_state()

_load_state(*action*)

_remove_state(*action*)

_find()

increase_arc_length()

decrease_arc_length()

add_entities_at_position(*checked=False*)

edit_selected(*_=False*)
Edits selected items.

remove_selected(*_=False*)
Removes selected items.

_get_selected_entity_names()

hide_selected_items(*checked=False*)
Hides selected items.

_hide_class(*action*)
Hides some class.

show_all_hidden_items(*checked=False*)
Shows all hidden items.

show_hidden_items(*action*)
Shows some hidden items.

prune_selected_items(*checked=False*)
Prunes selected items.

_prune_class(*action*)
Prunnes some class.

restore_all_pruned_items(*checked=False*)
Reinstates all pruned items.

restore_pruned_items(*action*)
Reinstates some pruned items.

select_graph_parameters(*checked=False*)

_set_graph_parameters(*parameters*)

_save_selected_positions(*checked=False*)

_save_all_positions(*checked=False*)

_save_positions(*items*)

_clear_selected_positions(*checked=False*)

_clear_all_positions(*checked=False*)

_clear_positions(*items*)

_select_bg_image(*_checked=False*)

set_bg_svg(*svg*)

get_bg_svg()

set_bg_rect(*rect*)

get_bg_rect()

clear_scene()

_no_zoom()

export_as_image(*_=False*)

_do_export_as_image(*file_path*)

_get_print_source(*scene=None*)

_print_scene(*printer*, *source*, *size*, *index=None*, *scene=None*)

_clone_scene()

_frames(*start*, *stop*, *step_len*, *buffer_path*, *cv2*)

export_as_video()

_do_export_as_video(*file_path*, *start*, *stop*, *step_len*, *fps*, *cv2*)

set_cross_hairs_items(*entity_class*, *cross_hairs_items*)

Sets 'cross_hairs' items for connecting entities.

Parameters

- **entity_class** (*dict*) –
- **cross_hairs_items** (*list(QGraphicsItems)*) –

clear_cross_hairs_items()

_cross_hairs_has_valid_target()

mousePressEvent(*event*)

Handles relationship creation if one it's in process.

mouseMoveEvent(*event*)

Updates the hovered object item if we're in entity creation mode.

_update_cross_hairs_pos(*pos*)

Updates the hovered object item and sets the 'cross_hairs' icon accordingly.

Parameters

pos (*QPoint*) – the desired position in view coordinates

mouseReleaseEvent(*event*)

Reestablish scroll hand drag mode.

keyPressEvent(*event*)

Aborts relationship creation if user presses ESC.

contextMenuEvent(*e*)

Shows context menu.

Parameters
e (*QContextMenuEvent*) – Context menu event

_compute_max_zoom()

_use_smooth_zoom()

_zoom(*factor*)

apply_zoom()

wheelEvent(*event*)

Zooms in/out. If user has pressed the shift key, rotates instead.

Parameters
event (*QWheelEvent*) – Mouse wheel event

_handle_rotation_time_line_advanced(*pos*)

Performs rotation whenever the smooth rotation time line advances.

_rotate(*angle*)

rotate_clockwise()

Performs a rotate clockwise with fixed angle.

rotate_anticlockwise()

Performs a rotate anticlockwise with fixed angle.

spinetoolbox.spine_db_editor.widgets.custom_qtableview

Custom QTableView classes that support copy-paste and the like.

Module Contents

Classes

StackedTableView	Base stacked view.
ParameterTableView	Base stacked view.
ParameterDefinitionTableView	Base stacked view.
ParameterValueTableView	Base stacked view.
EntityAlternativeTableView	Visualize entities and their alternatives.
PivotTableView	Custom QTableView class with pivot capabilities.
FrozenTableView	**param parent** parent widget
MetadataTableViewBase	Base for metadata and item metadata table views.
MetadataTableView	Table view for metadata.
ItemMetadataTableView	Table view for entity and parameter value metadata.

Functions

_set_data(index, new_value)	Updates (object or relationship) parameter_definition or value with newly edited data.

spinetoolbox.spine_db_editor.widgets.custom_qtableview.**_set_data**(*index*, *new_value*)

Updates (object or relationship) parameter_definition or value with newly edited data.

class spinetoolbox.spine_db_editor.widgets.custom_qtableview.**StackedTableView**(*parent*)

Bases: *spinetoolbox.widgets.custom_qtableview.AutoFilterCopyPasteTableView*

Base stacked view.

Parameters
parent (*QWidget*) – parent widget

_COLUMN_SIZE_HINTS

_EXPECTED_COLUMN_COUNT: int

connect_spine_db_editor(*spine_db_editor*)

Connects a Spine db editor to work with this view.

Parameters
spine_db_editor (SpineDBEditor) –

_make_delegate(*column_name*, *delegate_class*)

Creates a delegate for the given column and returns it.

Parameters

- **column_name** (*str*) –
- **delegate_class** (TableDelegate) –

Returns
TableDelegate

create_delegates()

Creates delegates for this view

populate_context_menu()

Creates a context menu for this view.

contextMenuEvent(*event*)

Shows context menu.

Parameters
event (*QContextMenuEvent*) –

_selected_rows_per_column()

Computes selected rows per column.

Returns
Mapping columns to selected rows in that column.

Return type
dict

filter_by_selection(*checked=False*)

filter_excluding_selection(*checked=False*)

remove_selected()

Removes selected indexes.

_refresh_copy_paste_actions(*_*, *__*)

Enables or disables copy and paste actions.

_initial_column_size(*column*)

_set_column_resize_modes(*old_column_count*, *new_column_count*)

class spinetoolbox.spine_db_editor.widgets.custom_qtableview.**ParameterTableView**(*parent*)

Bases: *StackedTableView*

Base stacked view.

Parameters
parent (*QWidget*) – parent widget

value_column_header: str

Either "default_value" or "value". Used to identify the value column for advanced editing and plotting.

populate_context_menu()

Creates a context menu for this view.

contextMenuEvent(*event*)

Shows context menu.

Parameters
event (*QContextMenuEvent*) –

open_in_editor()

Opens the current index in a parameter_value editor using the connected Spine db editor.

plot(*checked=False*)

Plots current index.

abstract _plot_selection(*selection*, *plot_widget=None*)

Adds selected indexes to existing plot or creates a new plot window.

Parameters

- **selection** (*Iterable of QModelIndex*) – a list of QModelIndex objects for plotting
- **plot_widget** (*PlotWidget, optional*) – an existing plot widget to draw into or None to create a new widget

Returns
a PlotWidget object

Return type
PlotWidget

plot_in_window(*action*)

Plots current index in the window given by action's name.

class spinetoolbox.spine_db_editor.widgets.custom_qtableview.**ParameterDefinitionTableView**(*parent*)

Bases: *ParameterTableView*

Base stacked view.

Parameters
parent (*QWidget*) – parent widget

value_column_header = 'default_value'

_EXPECTED_COLUMN_COUNT = 6

_COLUMN_SIZE_HINTS

create_delegates()

Creates delegates for this view

_plot_selection(*selection*, *plot_widget=None*)

See base class

class spinetoolbox.spine_db_editor.widgets.custom_qtableview.**ParameterValueTableView**(*parent*)

Bases: *ParameterTableView*

Base stacked view.

Parameters
parent (*QWidget*) – parent widget

property _pk_fields

value_column_header = 'value'

_COLUMN_SIZE_HINTS

_EXPECTED_COLUMN_COUNT = 6

connect_spine_db_editor(*spine_db_editor*)

Connects a Spine db editor to work with this view.

Parameters
spine_db_editor (SpineDBEditor) –

create_delegates()

Creates delegates for this view

_update_pinned_values(*_selected*, *_deselected*)

_make_pinned_value(*index*)

_plot_selection(*selection*, *plot_widget=None*)

See base class.

class spinetoolbox.spine_db_editor.widgets.custom_qtableview.**EntityAlternativeTableView**(*parent*)

Bases: *StackedTableView*

Visualize entities and their alternatives.

Parameters
parent (*QWidget*) – parent widget

_EXPECTED_COLUMN_COUNT = 5

_COLUMN_SIZE_HINTS

create_delegates()

Creates delegates for this view

class spinetoolbox.spine_db_editor.widgets.custom_qtableview.**PivotTableView**(*parent=None*)

Bases: *spinetoolbox.widgets.custom_qtableview.CopyPasteTableView*

Custom QTableView class with pivot capabilities.

Uses 'contexts' to provide different UI elements (table headers, context menus,...) depending on what data the pivot table currently contains.

Parameters
parent (*QWidget, optional*) – parent widget

class _ContextBase(*view*, *db_editor*, *horizontal_header*, *vertical_header*)

Base class for pivot table view's contexts.

Parameters

- **view** (PivotTableView) – parent view
- **db_editor** (SpineDBEditor) – database editor
- **horizontal_header** (*QHeaderView*) – horizontal header
- **vertical_header** (*QHeaderView*) – vertical header

_REMOVE_ENTITY = 'Remove entities'

_REMOVE_PARAMETER = 'Remove parameter definitions'

_REMOVE_ALTERNATIVE = 'Remove alternatives'

_REMOVE_SCENARIO = 'Remove scenarios'

_DUPLICATE_SCENARIO = 'Duplicate scenario'

_clear_selection_lists()

Clears cached selected index lists.

abstract populate_context_menu()

Generates context menu.

_refresh_selected_indexes()

Caches selected index lists.

remove_alternatives()

Removes selected alternatives from the database.

show_context_menu(*position*)

Shows the context menu.

_to_selection_lists(*index*)

Caches given index to corresponding selected index list.
Parameters
index (*QModelIndex*) – index to cache

abstract _update_actions_availability()

Enables/disables context menu entries before the menu is shown.

class _EntityContextBase(*view*, *db_editor*, *horizontal_header*, *vertical_header*)

Bases: *PivotTableView._ContextBase*

Base class for contexts that contain entities and entity classes.

Parameters

- **view** (PivotTableView) – parent view
- **db_editor** (SpineDBEditor) – database editor
- **horizontal_header** (*QHeaderView*) – horizontal header
- **vertical_header** (*QHeaderView*) – vertical header

_clear_selection_lists()

See base class.

abstract populate_context_menu()

See base class.

remove_entities()

Removes selected entities from the database.

abstract _update_actions_availability()

See base class.

class _ParameterValueContext(*view*, *db_editor*)

Bases: *PivotTableView._EntityContextBase*

Context for showing parameter values in the pivot table.

Parameters

- **view** (PivotTableView) – parent view
- **db_editor** (SpineDBEditor) – database editor

_clear_selection_lists()

See base class.

populate_context_menu()

See base class.

open_in_editor()

Opens the parameter value editor for the first selected cell.

plot()

Plots the selected cells.

_plot_in_window(*action*)

Plots the selected cells in an existing window.

remove_parameters()

Removes selected parameter definitions from the database.

remove_values()

Removes selected parameter values from the database.

show_context_menu(*position*)

Shows the context menu.

_to_selection_lists(*index*)

See base class.

_update_actions_availability()

See base class.

class _IndexExpansionContext(*view*, *db_editor*)

Bases: *PivotTableView._ParameterValueContext*

Context for expanded parameter values

Parameters

- **view** (PivotTableView) – parent view
- **db_editor** (SpineDBEditor) – database editor

class _ElementContext(*view*, *db_editor*)

Bases: *PivotTableView._EntityContextBase*

Context for presenting relationships in the pivot table.

Parameters

- **view** (PivotTableView) – parent view
- **db_editor** (SpineDBEditor) – database editor

populate_context_menu()

See base class.

_update_actions_availability()

See base class.

class _ScenarioAlternativeContext(*view*, *db_editor*)

Bases: *PivotTableView._ContextBase*

Context for presenting scenarios and alternatives

Parameters

- **view** (PivotTableView) – parent view
- **db_editor** (SpineDBEditor) – database editor

_clear_selection_lists()

See base class.

populate_context_menu()

See base class.

remove_scenarios()

Removes selected scenarios from the database.

duplicate_scenario()

Duplicates current scenario in the database.

_to_selection_lists(*index*)

See base class.

_update_actions_availability()

See base class.

_open_scenario_generator()

Opens the scenario generator dialog.

_toggle_checked_state()

Toggles the checked state of selected alternatives.

property source_model

property db_mngr

header_changed

connect_spine_db_editor(*spine_db_editor*)

_change_context()

Changes the UI engine according to pivot model type.

contextMenuEvent(*event*)

Shows context menu.

Parameters
event (*QContextMenuEvent*) –

setModel(*model*)

_synch_selection_with_header_tables(*selected*, *deselected*)

indexWidget(*proxy_index*)

setIndexWidget(*proxy_index*, *widget*)

setHorizontalHeader(*horizontal_header*)

setVerticalHeader(*vertical_header*)

resizeEvent(*ev*)

sizeHintForColumn(*column*)

_fetch_more_visible()

_update_header_tables()

_update_section_width(*logical_index*, *_old_size*, *new_size*)

_update_section_height(*logical_index*, *_old_size*, *new_size*)

_update_header_tables_geometry()

_refresh_copy_paste_actions(*_*, *__*)

class spinetoolbox.spine_db_editor.widgets.custom_qtableview.FrozenTableView(*parent=None*)

Bases: PySide6.QtWidgets.QTableView

Parameters
parent (*QWidget*) – parent widget

property area

header_dropped

dragEnterEvent(*event*)

dragMoveEvent(*event*)

dropEvent(*event*)

class spinetoolbox.spine_db_editor.widgets.custom_qtableview.**MetadataTableViewBase**(*parent*)

Bases: *spinetoolbox.widgets.custom_qtableview.CopyPasteTableView*

Base for metadata and item metadata table views.

Parameters
parent (*QWidget, optional*) – parent widget

connect_spine_db_editor(*db_editor*)

Finishes view's initialization.

Parameters
db_editor (SpineDBEditor) – database editor instance

contextMenuEvent(*event*)

_remove_selected()

Removes selected rows from view's model.

_enable_delegates(*db_editor*)

Creates delegates for this view

Parameters
db_editor (SpineDBEditor) – database editor

_populate_context_menu()

Fills context menu with actions.

_set_model_data(*index*, *value*)

Sets model data.

Parameters

- **index** (*QModelIndex*) – model index to set
- **value** (*str*) – value

_refresh_copy_paste_actions()

_set_horizontal_header_resize_modes(*old_column_count*, *new_column_count*)

class spinetoolbox.spine_db_editor.widgets.custom_qtableview.**MetadataTableView**(*parent*)

Bases: *MetadataTableViewBase*

Table view for metadata.

Parameters
parent (*QWidget, optional*) – parent widget

_enable_delegates(*db_editor*)

See base class.

class spinetoolbox.spine_db_editor.widgets.custom_qtableview.**ItemMetadataTableView**(*parent*)

Bases: *MetadataTableViewBase*

Table view for entity and parameter value metadata.

Parameters
parent (*QWidget*) – parent widget

set_models(*item_metadata_model*, *metadata_model*)

Sets models.

Parameters

- **item_metadata_model** (*ItemMetadataModel*) – item metadata model
- **metadata_model** (MetadataTableModel) – metadata model

_enable_delegates(*db_editor*)

See base class

spinetoolbox.spine_db_editor.widgets.custom_qtreeview

Classes for custom QTreeViews and QTreeWidgets.

Module Contents

Classes

EntityTreeView	Tree view for entity classes and entities.
ItemTreeView	Base class for all non-entity tree views.
AlternativeTreeView	Custom QTreeView for the alternative tree in SpineDBEditor.
ScenarioTreeView	Custom QTreeView for the scenario tree in SpineDBEditor.
ParameterValueListTreeView	Custom QTreeView class for parameter_value_list in SpineDBEditor.

class spinetoolbox.spine_db_editor.widgets.custom_qtreeview.**EntityTreeView**(*parent*)

Bases: *spinetoolbox.widgets.custom_qtreeview.CopyPasteTreeView*

Tree view for entity classes and entities.

Parameters
parent (*QWidget*) – parent widget

tree_selection_changed

reset()

connect_spine_db_editor(*spine_db_editor*)

Connects a Spine db editor to work with this view.

Parameters
spine_db_editor (SpineDBEditor) –

_add_middle_actions()

_create_context_menu()

Creates a context menu for this view.

toggle_hide_empty_classes()

edit(*index*, *trigger*, *event*)

Edit all selected items.

connect_signals()

Connects signals.

rowsInserted(*parent*, *start*, *end*)

rowsRemoved(*parent*, *start*, *end*)

setModel(*model*)

_fetch_more_visible()

verticalScrollbarValueChanged(*value*)

_handle_selection_changed(*selected*, *deselected*)

Classifies selection by item type and emits signal.

_refresh_selected_indexes()

clear_any_selections()

Clears the selection if any.

fully_expand()

Expands selected indexes and all their children.

fully_collapse()

Collapses selected indexes and all their children.

export_selected()

Exports data from selected indexes using the connected Spine db editor.

remove_selected()

Removes selected indexes using the connected Spine db editor.

contextMenuEvent(*event*)

Shows context menu.

Parameters
event (*QContextMenuEvent*) –

mousePressEvent(*event*)

Overrides selection behaviour if the user has selected sticky selection in Settings. If sticky selection is enabled, multiple-selection is enabled when selecting items in the Object tree. Pressing the Ctrl-button down, enables single selection.

Parameters
event (*QMouseEvent*) –

update_actions_availability()

Updates the visible property of actions according to whether or not they apply to given item.

edit_selected()

Edits all selected indexes using the connected Spine db editor.

add_entity_classes()

add_entities()

find_next_entity()

Finds the next occurrence of the relationship at the current index and expands it.

_do_find_next_entity()

duplicate_entity()

Duplicates the object at the current index using the connected Spine db editor.

add_entity_group()

manage_elements()

manage_members()

select_superclass()

class spinetoolbox.spine_db_editor.widgets.custom_qtreeview.**ItemTreeView**(*parent*)

Bases: *spinetoolbox.widgets.custom_qtreeview.CopyPasteTreeView*

Base class for all non-entity tree views.

Parameters
parent (*QWidget*) – parent widget

rowsInserted(*parent*, *start*, *end*)

connect_signals()

Connects signals.

abstract remove_selected()

Removes items selected in the view.

abstract update_actions_availability(*item*)

Updates the visible property of actions according to whether or not they apply to given item.

connect_spine_db_editor(*spine_db_editor*)

Prepares the view to work with the DB editor.

Parameters
spine_db_editor (SpineDBEditor) – editor instance

populate_context_menu()

Creates a context menu for this view.

contextMenuEvent(*event*)

Shows context menu.

Parameters
event (*QContextMenuEvent*) –

_refresh_copy_paste_actions(_, __)

Refreshes copy and paste actions enabled state.

class spinetoolbox.spine_db_editor.widgets.custom_qtreeview.**AlternativeTreeView**(*parent*)

Bases: *ItemTreeView*

Custom QTreeView for the alternative tree in SpineDBEditor.

Parameters
parent (*QWidget*) – parent widget

property selected_alternative_ids

alternative_selection_changed

reset()

connect_signals()
Connects signals.

connect_spine_db_editor(*spine_db_editor*)
see base class

populate_context_menu()
See base class.

_db_map_alt_ids_from_selection(*selection*)
Gather alternative ids per database map from selection.

Parameters
selection (*QItemSelection*) – selection

Returns
mapping from database map to set of alternative ids

Return type
dict

_handle_selection_changed(*selected*, *deselected*)
Emits alternative_selection_changed with the current selection.

remove_selected()
See base class.

update_actions_availability(*item*)
See base class.

_open_scenario_generator()
Opens the scenario generator dialog.

can_copy()
See base class.

can_paste()
See base class.

copy()
See base class.

paste()
Pastes alternatives from clipboard to the tree.

This makes sense only when pasting alternatives from one database to another.

class spinetoolbox.spine_db_editor.widgets.custom_qtreeview.**ScenarioTreeView**(*parent*)
Bases: *ItemTreeView*

Custom QTreeView for the scenario tree in SpineDBEditor.

Parameters
parent (*QWidget*) – parent widget

property selected_alternative_ids

scenario_selection_changed

reset()

connect_signals()
Connects signals.

connect_spine_db_editor(*spine_db_editor*)
see base class

populate_context_menu()
See base class.

_db_map_alternative_ids_from_selection(*selection*)
Collects database maps and alternative ids within given selection.

Parameters
selection (*Sequence of QModelIndex*) – selection indices

Returns
mapping from database map to set of alternative ids

Return type
dict

_handle_selection_changed(*selected*, *deselected*)
Emits scenario_selection_changed with the current selection.

remove_selected()
See base class.

dragMoveEvent(*event*)

dragEnterEvent(*event*)

update_actions_availability(*item*)
See base class

copy()
See base class.

can_paste()
See base class.

paste()
Pastes alternatives and scenarios from clipboard to the tree.

_duplicate_scenario()
Duplicates selected scenarios.

class spinetoolbox.spine_db_editor.widgets.custom_qtreeview.**ParameterValueListTreeView**(*parent*)
Bases: *ItemTreeView*

Custom QTreeView class for parameter_value_list in SpineDBEditor.

Parameters
parent (*QWidget*) – parent widget

connect_spine_db_editor(*spine_db_editor*)

see base class

populate_context_menu()

Creates a context menu for this view.

update_actions_availability(*item*)

See base class.

open_in_editor()

Opens the parameter_value editor for the first selected cell.

remove_selected()

See base class.

spinetoolbox.spine_db_editor.widgets.custom_qwidgets

Custom QWidgets.

Module Contents

Classes

OpenFileButton	A button to open files or show them in the folder.
OpenSQLiteFileButton	A button to open sqlite files, show them in the folder, or add them to the project.
ShootingLabel	
ProgressBarWidget	
TimeLineWidget	
LegendWidget	
ExportAsVideoDialog	

class spinetoolbox.spine_db_editor.widgets.custom_qwidgets.**OpenFileButton**(*file_path*, *progress*, *db_editor*)

Bases: PySide6.QtWidgets.QWidget

A button to open files or show them in the folder.

set_progress(*progress*)

open_file(*checked=False*)

open_containing_folder(*checked=False*)

class spinetoolbox.spine_db_editor.widgets.custom_qwidgets.**OpenSQLiteFileButton**(*file_path*, *progress*, *db_editor*)

Bases: *OpenFileButton*

A button to open sqlite files, show them in the folder, or add them to the project.

open_file(*checked=False*)

class spinetoolbox.spine_db_editor.widgets.custom_qwidgets.**ShootingLabel**(*origin*, *destination*, *parent=None*, *duration=1200*)

Bases: PySide6.QtWidgets.QLabel

_handle_value_changed(*value*)

show()

class spinetoolbox.spine_db_editor.widgets.custom_qwidgets.**ProgressBarWidget**(*parent=None*)

Bases: PySide6.QtWidgets.QWidget

set_layout_generator(*layout_generator*)

paintEvent(*event*)

class spinetoolbox.spine_db_editor.widgets.custom_qwidgets.**TimeLineWidget**(*parent=None*)

Bases: PySide6.QtWidgets.QWidget

index_changed

_update_step_len(*value*)

_refresh_button_icon()

_play_pause()

_update_playback()

_handle_value_changed(*value*)

set_index_range(*min_index*, *max_index*)

get_index_range()

class spinetoolbox.spine_db_editor.widgets.custom_qwidgets.**LegendWidget**(*parent=None*)

Bases: PySide6.QtWidgets.QWidget

_BASE_HEIGHT = 30

_SPACING = 6

row_count()

set_legend(*legend*)

static _paint_color_bar(*painter*, *cell*)

static _paint_volume_bar(*painter*, *cell*)

paintEvent(*ev*)

paint(*painter*, *rect*)

class spinetoolbox.spine_db_editor.widgets.custom_qwidgets.**ExportAsVideoDialog**(*start*, *stop*, *parent=None*)

Bases: PySide6.QtWidgets.QDialog

_handle_start_dt_changed(*start_dt*)

_handle_stop_dt_changed(*stop_dt*)

selections()

spinetoolbox.spine_db_editor.widgets.edit_or_remove_items_dialogs

Classes for custom QDialogs to edit items in databases.

Module Contents

Classes

EditOrRemoveItemsDialog	A dialog with a CopyPasteTableView and a QDialogButtonBox. Base class for all
EditEntityClassesDialog	A dialog to query user's preferences for updating entity classes.
EditEntitiesDialog	A dialog to query user's preferences for updating entities.
RemoveEntitiesDialog	A dialog to query user's preferences for removing tree items.
SelectSuperclassDialog	Provides a method to retrieve entity classes for AddEntitiesDialog and AddEntityClassesDialog.

class spinetoolbox.spine_db_editor.widgets.edit_or_remove_items_dialogs.**EditOrRemoveItemsDialog**(*parent*, *db_mngr*)

Bases: *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.ManageItemsDialog*

A dialog with a CopyPasteTableView and a QDialogButtonBox. Base class for all dialogs to query user's preferences for adding/editing/managing data items.

Parameters

- **parent** (SpineDBEditor) – data store widget
- **db_mngr** (SpineDBManager) –

all_databases(*row*)

Returns a list of db names available for a given row. Used by delegates.

class spinetoolbox.spine_db_editor.widgets.edit_or_remove_items_dialogs.**EditEntityClassesDialog**(*parent*, *db_mngr*, *selected*)

Bases: *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.ShowIconColorEditorMixin*, *EditOrRemoveItemsDialog*

A dialog to query user's preferences for updating entity classes.

Parameters

- **parent** (SpineDBEditor) – data store widget
- **db_mngr** (SpineDBManager) – the manager to do the update
- **selected** (*set*) – set of EntityClassItem instances to edit

connect_signals()

Connect signals to slots.

accept()

Collect info from dialog and try to update items.

class spinetoolbox.spine_db_editor.widgets.edit_or_remove_items_dialogs.**EditEntitiesDialog**(*parent*, *db_mngr*, *selected*, *class_key*)

Bases: *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.GetEntityClassesMixin*, *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.GetEntitiesMixin*, *EditOrRemoveItemsDialog*

A dialog to query user's preferences for updating entities.

Parameters

- **parent** (SpineDBEditor) – data store widget
- **db_mngr** (SpineDBManager) – the manager to do the update
- **selected** (*set*) – set of EntityItem instances to edit
- **class_key** (*tuple*) – for identifying the entity class

accept()

Collect info from dialog and try to update items.

class spinetoolbox.spine_db_editor.widgets.edit_or_remove_items_dialogs.**RemoveEntitiesDialog**(*parent*, *db_mngr*, *selected*)

Bases: *EditOrRemoveItemsDialog*

A dialog to query user's preferences for removing tree items.

Parameters

- **parent** (SpineDBEditor) – data store widget
- **db_mngr** (SpineDBManager) – the manager to do the removal
- **selected** (*dict*) – maps item type (class) to instances

accept()

Collect info from dialog and try to remove items.

class spinetoolbox.spine_db_editor.widgets.edit_or_remove_items_dialogs.**SelectSuperclassDialog**(*parent*, *entity_class_*, *db_mngr*, **db_maps*)

Bases: *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.GetEntityClassesMixin*, *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.DialogWithButtons*

Provides a method to retrieve entity classes for AddEntitiesDialog and AddEntityClassesDialog.

Parameters

- **parent** (SpineDBEditor) – data store widget
- **db_mngr** (SpineDBManager) –

_populate_layout()

accept()

spinetoolbox.spine_db_editor.widgets.element_name_list_editor

Contains the ElementNameListEditor class.

Module Contents

Classes

SearchBarDelegate	A custom delegate to use with ElementNameListEditor.
ElementNameListEditor	A dialog to select the element name list for an entity using Google-like search bars.

class spinetoolbox.spine_db_editor.widgets.element_name_list_editor.**SearchBarDelegate**

Bases: PySide6.QtWidgets.QItemDelegate

A custom delegate to use with ElementNameListEditor.

data_committed

setModelData(*editor*, *model*, *index*)

createEditor(*parent*, *option*, *index*)

updateEditorGeometry(*editor*, *option*, *index*)

close_editor(*editor*, *index*, *model*)

eventFilter(*editor*, *event*)

class spinetoolbox.spine_db_editor.widgets.element_name_list_editor.**ElementNameListEditor**(*parent*, *index*, *entity_class_names*, *entity_byname_lists*, *current_element_by*

Bases: *spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.ManageItemsDialog*

A dialog to select the element name list for an entity using Google-like search bars.

Parameters

- **parent** (SpineDBEditor) –
- **index** (*QModelIndex*) –
- **entity_class_names** (*list*) – string entity_class names
- **entity_byname_lists** (*list*) – lists of string entity names
- **current_element_byname_list** (*list*) –

init_model(*entity_class_names*, *entity_byname_lists*, *current_element_byname_list*)

accept()

spinetoolbox.spine_db_editor.widgets.graph_layout_generator

Contains the GraphLayoutGeneratorRunnable class.

Module Contents

Classes

GraphLayoutGeneratorRunnable	Computes the layout for the Entity Graph View.

class spinetoolbox.spine_db_editor.widgets.graph_layout_generator.**GraphLayoutGeneratorRunnable**(*identifier*, *vertex_count*, *src_inds=(*, *dst_inds=(*, *spread=0*, *heavy_pos*, *max_iters=*, *weight_exp*, *2*)

Bases: PySide6.QtCore.QRunnable

Computes the layout for the Entity Graph View.

class Signals

Bases: PySide6.QtCore.QObject

finished

layout_available

progressed

stop(*_checked=False*)

set_show_previews(*checked*)

_is_stopped()

_layout_progressed(*iteration*)

_layout_available(*x*, *y*)

_preview_available(*x*, *y*)

run()

spinetoolbox.spine_db_editor.widgets.graph_view_mixin

Contains the GraphViewMixin class.

Module Contents

Classes

GraphViewMixin	Provides the graph view for the DB editor.
_Offset	

Functions

_min_value(pv)
_max_value(pv)
_get_value(pv, index)
_min_max(pvs)
_min_max_indexes(pvs)

spinetoolbox.spine_db_editor.widgets.graph_view_mixin.**_min_value**(*pv*)

spinetoolbox.spine_db_editor.widgets.graph_view_mixin.**_max_value**(*pv*)

spinetoolbox.spine_db_editor.widgets.graph_view_mixin.**_get_value**(*pv*, *index*)

spinetoolbox.spine_db_editor.widgets.graph_view_mixin.**_min_max**(*pvs*)

spinetoolbox.spine_db_editor.widgets.graph_view_mixin.**_min_max_indexes**(*pvs*)

class spinetoolbox.spine_db_editor.widgets.graph_view_mixin.**GraphViewMixin**(**args*, ***kwargs*)

Provides the graph view for the DB editor.

NOT_SPECIFIED

_VERTEX_EXTENT = 64

_ARC_WIDTH

_ARC_LENGTH_HINT

_update_time_line_index(*index*)

_graph_fetch_more_later(*entity=True*, *parameter_value=True*)

_graph_fetch_more(*entity=True*, *parameter_value=True*)

_graph_fetch_more_parameter_value()

_graph_fetch_more_entity()

init_models()

connect_signals()

Connects signals.

_refresh_icons(*item_type*, *db_map_data*)

Runs when entity classes are added or updated in the db. Refreshes icons of entities in graph.

Parameters

db_map_data (*dict*) – list of dictionary-items keyed by DiffDatabaseMapping instance.

_all_pruned_db_map_entity_ids()

_accepts_entity_item(*item*, *db_map*)

_graph_handle_entities_added(*db_map_data*)

Runs when entities are added to the db. Adds the new entities to the graph if needed.

Parameters

db_map_data (*dict*) – list of dictionary-items keyed by DiffDatabaseMapping instance.

_graph_handle_entities_removed(*db_map_data*)

Runs when entities are removed from the db. Rebuilds graph if needed.

Parameters

db_map_data (*dict*) – list of dictionary-items keyed by DiffDatabaseMapping instance.

_graph_handle_entities_updated(*db_map_data*)

Runs when entities are updated in the db.

Parameters

db_map_data (*dict*) – list of dictionary-items keyed by DiffDatabaseMapping instance.

_db_map_ids_by_key(*db_map_data*)

add_db_map_ids_to_items(*db_map_data*)

Goes through entity items and adds the corresponding db_map ids. This could mean either restoring removed (db_map, id) tuples previously removed, or adding new (db_map, id) tuples.

Parameters

db_map_data (*dict(DiffDatabaseMapping, list)*) – List of added items keyed by db_map

Returns
tuples (db_map, id) that didn't match any item in the view.

Return type
list

_graph_handle_parameter_values_added(*db_map_data*)

polish_items()

_update_property_pvs()

_handle_entity_graph_visibility_changed(*visible*)

_handle_entity_tree_selection_changed_in_graph(*selected*)
Stores the given selection of entity tree indexes and builds graph.

expand_graph(*db_map_entity_ids*)

collapse_graph(*db_map_entity_ids*)

prune_graph(*key*, *db_map_entity_ids*)

restore_graph(*key=None*)

_get_db_map_graph_data()

save_graph_data(*name*)

overwrite_graph_data(*db_map_graph_data*)

get_db_map_graph_data_by_name()

load_graph_data(*db_map_graph_data*)

remove_graph_data(*name*)

rebuild_graph(*_checked=False*)

build_graph(*persistent=False*)
Builds graph from current selection of items.

Parameters
persistent (*bool, optional*) – If True, elements in the current graph (if any) retain their position in the new one.

_refresh_graph()

_stop_layout_generators()

_complete_graph(*layout_gen_id*, *x*, *y*)

Parameters

- **layout_gen_id** (*object*) –
- **x** (*list*) – Horizontal coordinates
- **y** (*list*) – Vertical coordinates

_update_selected_item_type_db_map_ids(*selected_tree_inds*)
Upsates the dict mapping item type to db_map to selected ids.

_get_db_map_entities_for_graph()

_update_graph_data()

Updates data for graph according to selection in trees.

get_entity_key(*db_map_entity_id*)

_update_entity_element_inds(*db_map_element_id_lists*)

_get_pv(*db_map*, *entity_id*, *pname*)

get_item_name(*db_map*, *entity_id*)

get_item_color(*db_map*, *entity_id*, *time_line_index*)

get_arc_width(*db_map*, *entity_id*, *time_line_index*)

get_vertex_radius(*db_map*, *entity_id*, *time_line_index*)

_get_item_property(*db_map*, *entity_id*, *pname*, *time_line_index*)

Returns a tuple of (min_value, value, max_value) for given entity and property. Returns self.NOT_SPECIFIED if the property is not defined for the entity. Returns None if the property is not defined for *any* entity.

Returns
tuple or None

_get_fixed_pos(*db_map*, *entity_id*)

_make_layout_generator()

Returns a layout generator for the current graph.

Returns
GraphLayoutGeneratorRunnable

static convert_position(*x*, *y*)

_get_entity_offset(*db_map_entity_ids*)

_make_new_items(*x*, *y*)

Makes new items for the graph.

Parameters

- **x** (*list*) –
- **y** (*list*) –

Returns
True if graph contains any items after the operation, False otherwise

Return type
bool

_add_new_items()

start_connecting_entities(*db_map*, *entity_class*, *ent_item*)

Starts connecting entites with the given entity item.

Parameters

- **db_map** (*DiffDatabaseMapping*) –

- **entity_class** (*dict*) –
- **ent_item** (*..graphics_items.EntityItem*) –

finalize_connecting_entities(*entity_class*, **entity_items*)

Tries to add multi dimensional entity with the given entity items as elements.

Parameters

- **entity_class** (*dict*) –
- **entity_items** (*..graphics_items.EntityItem*) –

_do_finalize_connecting_entities(*dialog*, *element_items*)

add_entities_at_position(*pos*)

_do_add_entites_at_pos(*dialog*, *x*, *y*)

_add_entities_from_dialog(*dialog*)

get_save_file_path(*group*, *caption*, *filters*)

get_open_file_path(*group*, *caption*, *filters*)

closeEvent(*event*)

Handle close window.

Parameters

event (*QCloseEvent*) – Closing event

class spinetoolbox.spine_db_editor.widgets.graph_view_mixin.**_Offset**(*all_offsets*)

value()

spinetoolbox.spine_db_editor.widgets.item_metadata_editor

Contains machinery to deal with item metadata editor.

Module Contents

Classes

ItemMetadataEditor	A DB editor helper class that manages entity and parameter value metadata editor.

class spinetoolbox.spine_db_editor.widgets.item_metadata_editor.**ItemMetadataEditor**(*item_metadata_table_view*, *db_editor*, *metadata_editor*, *db_mngr*)

A DB editor helper class that manages entity and parameter value metadata editor.

Parameters

- **item_metadata_table_view** (ItemMetadataTableView) – editor's view

- **db_editor** (SpineDBEditor) – database editor
- **metadata_editor** (MetadataEditor) – metadata editor
- **db_mngr** (SpineDBManager) – database manager

connect_signals(*ui*)

Connects user interface signals.

Parameters
ui (Ui_MainWindow) – DB editor's user interface

init_models(*db_maps*)

Initializes editor's models.

Parameters
db_maps (*Iterable of DiffDatabaseMapping*) – database mappings

_reload_entity_metadata(*current_index*, *previous_index*)

Loads entity metadata for selected object or relationship.

Parameters

- **current_index** (*QModelIndex*) – currently selected index in object/relationship tree
- **previous_index** (*QModelIndex*) – unused

_reload_value_metadata(*current_index*, *previous_index*)

Loads parameter value metadata for selected value.

Parameters

- **current_index** (*QModelIndex*) – currently selected index in object/relationship parameter value table
- **previous_index** (*QModelIndex*) – unused

spinetoolbox.spine_db_editor.widgets.manage_items_dialogs

Classes for custom QDialogs to add, edit and remove database items.

Module Contents

Classes

DialogWithButtons	**param parent** data store widget
DialogWithTableAndButtons	**param parent** data store widget
ManageItemsDialog	A dialog with a CopyPasteTableView and a QDialogButtonBox. Base class for all
GetEntityClassesMixin	Provides a method to retrieve entity classes for AddEntitiesDialog and AddEntityClassesDialog.
GetEntitiesMixin	Provides a method to retrieve entities for AddEntitiesDialog and EditEntitiesDialog.
ShowIconColorEditorMixin	Provides methods to show an *IconColorEditor* upon request.

class spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.**DialogWithButtons**(*parent*, *db_mngr*)

Bases: PySide6.QtWidgets.QDialog

Parameters

- **parent** (SpineDBEditor) – data store widget
- **db_mngr** (SpineDBManager) –

showEvent(*ev*)

_populate_layout()

connect_signals()

Connect signals to slots.

class spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.**DialogWithTableAndButtons**(*parent*, *db_mngr*)

Bases: *DialogWithButtons*

Parameters

- **parent** (SpineDBEditor) – data store widget
- **db_mngr** (SpineDBManager) –

_populate_layout()

showEvent(*ev*)

abstract make_table_view()

resize_window_to_columns(*height=None*)

class spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.**ManageItemsDialog**(*parent*, *db_mngr*)

Bases: *DialogWithTableAndButtons*

A dialog with a CopyPasteTableView and a QDialogButtonBox. Base class for all dialogs to query user's preferences for adding/editing/managing data items.

Parameters

- **parent** (SpineDBEditor) – data store widget
- **db_mngr** (SpineDBManager) –

make_table_view()

connect_signals()

Connect signals to slots.

_handle_model_data_changed(*top_left*, *bottom_right*, *roles*)

Reimplement in subclasses to handle changes in model data.

set_model_data(*index*, *data*)

Update model data.

_handle_model_reset()

Resize columns and form.

class spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.**GetEntityClassesMixin**

Provides a method to retrieve entity classes for AddEntitiesDialog and AddEntityClassesDialog.

db_map_ent_cls_lookup()

db_map_ent_cls_lookup_by_name()

entity_class_name_list(*row*)

Return a list of entity class names present in all databases selected for given row. Used by *ManageEntityClassesDelegate*.

_entity_class_name_list_from_db_maps(**db_maps*)

class spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.**GetEntitiesMixin**(**args*, ***kwargs*)

Provides a method to retrieve entities for AddEntitiesDialog and EditEntitiesDialog.

property class_key

property dimension_name_list

property class_name

db_map_ent_lookup()

db_map_alt_id_lookup()

alternative_name_list(*row*)

Return a list of alternative names present in all databases selected for given row. Used by *ManageEntitiesDelegate*.

entity_name_list(*row*, *column*)

Return a list of entity names present in all databases selected for given row. Used by *ManageEntitiesDelegate*.

class spinetoolbox.spine_db_editor.widgets.manage_items_dialogs.**ShowIconColorEditorMixin**

Provides methods to show an *IconColorEditor* upon request.

show_icon_color_editor(*index*)

spinetoolbox.spine_db_editor.widgets.mass_select_items_dialogs

Classes for custom QDialogs to add edit and remove database items.

Module Contents

Classes

_SelectDatabases	A widget that shows checkboxes for each database.
MassSelectItemsDialog	A dialog to query a selection of dbs and items from the user.
MassRemoveItemsDialog	A dialog to query user's preferences for mass removing db items.
MassExportItemsDialog	A dialog to let users chose items for JSON export.

class spinetoolbox.spine_db_editor.widgets.mass_select_items_dialogs.**_SelectDatabases**(*db_maps*, *checked_states*, *parent*)

Bases: PySide6.QtWidgets.QWidget

A widget that shows checkboxes for each database.

Parameters

- **db_maps** (*tuple of DatabaseMapping*) – database maps
- **checked_states** (*dict, optional*) – mapping from item name to check state boolean
- **parent** (*QWidget*) – parent widget

checked_state_changed

checked_states()

Collects the checked states of databases.

Returns
mapping from database mapping to checked state boolean

Return type
dict

any_checked()

Checks if any of the checkboxes is checked.

Returns
True if any check box is checked, False otherwise

Return type
bool

class spinetoolbox.spine_db_editor.widgets.mass_select_items_dialogs.**MassSelectItemsDialog**(*parent*, *db_mngr*, **db_maps*, *stored_state*, *ok_button_text*)

Bases: *spinetoolbox.widgets.custom_qwidgets.SelectDatabaseItemsDialog*

A dialog to query a selection of dbs and items from the user.

Parameters

- **parent** (SpineDBEditor) – parent widget
- **db_mngr** (SpineDBManager) – database manager
- ***db_maps** – the dbs to select items from
- **stored_state** (*dict, Optional*) – widget's previous state
- **ok_button_text** (*str, optional*) – alternative label for the OK button

state_storing_requested

_handle_check_box_state_changed(*_checked*)

Enables or disables the OK button.

accept()

class spinetoolbox.spine_db_editor.widgets.mass_select_items_dialogs.**MassRemoveItemsDialog**(*parent*, *db_mngr*, **db_maps*, *stored_state=No*

Bases: *MassSelectItemsDialog*

A dialog to query user's preferences for mass removing db items.

Initialize class.

Parameters

- **parent** (SpineDBEditor) –
- **db_mngr** (SpineDBManager) –
- **db_maps** (*DiffDatabaseMapping*) – the dbs to select items from
- **stored_state** (*dict, Optional*) – widget's previous state

accept()

class spinetoolbox.spine_db_editor.widgets.mass_select_items_dialogs.**MassExportItemsDialog**(*parent*, *db_mngr*, **db_maps*, *stored_state=No*

Bases: *MassSelectItemsDialog*

A dialog to let users chose items for JSON export.

Parameters

- **parent** (SpineDBEditor) –
- **db_mngr** (SpineDBManager) –
- **db_maps** (*DiffDatabaseMapping*) – the dbs to select items from

- **stored_state** (*dict, Optional*) – widget's previous state

_warn_checked_non_data_items = False

data_submitted

accept()

spinetoolbox.spine_db_editor.widgets.metadata_editor

Contains machinery to deal with metadata editor.

Module Contents

Classes

MetadataEditor	A DB editor helper class that manages metadata editor.

class spinetoolbox.spine_db_editor.widgets.metadata_editor.**MetadataEditor**(*metadata_table_view*, *db_editor*, *db_mngr*)

A DB editor helper class that manages metadata editor.

Parameters

- **metadata_table_view** (MetadataTableView) – editor's view
- **db_editor** (SpineDBEditor) – database editor
- **db_mngr** (SpineDBManager) – database manager

connect_signals(*ui*)

Connects user interface signals.

Parameters
ui (Ui_MainWindow) – DB editor's user interface

init_models(*db_maps*)

Initializes editor's models.

Parameters
db_maps (*Iterable of DiffDatabaseMapping*) – database mappings

metadata_model()

Returns metadata model.

Returns
model

Return type
MetadataModel

spinetoolbox.spine_db_editor.widgets.multi_spine_db_editor

Contains the MultiSpineDBEditor class.

Module Contents

Classes

MultiSpineDBEditor	Database editor's tabbed main window.
_CustomStatusBar	

class spinetoolbox.spine_db_editor.widgets.multi_spine_db_editor.**MultiSpineDBEditor**(*db_mngr*, *db_url_codenames=None*)

Bases: *spinetoolbox.widgets.multi_tab_window.MultiTabWindow*

Database editor's tabbed main window.

Parameters

- **db_mngr** (SpineDBManager) – database manager
- **db_url_codenames** (*dict, optional*) – mapping from database URL to its codename

_make_other()

Creates a new MultiTabWindow of this type.

Returns
new MultiTabWindow

Return type
MultiTabWindow

_connect_tab_signals(*tab*)

Connects Spine Db editor window (tab) signals.

Parameters
tab (SpineDBEditor) – Spine Db editor window

Returns
True if ok, False otherwise

Return type
bool

_disconnect_tab_signals(*index*)

Disconnects signals of Spine Db editor window (tab) in given index.

Parameters
index (*int*) – Tab index

Returns
True if ok, False otherwise

Return type
bool

_make_new_tab(*db_url_codenames=None*, *window=False*)

Makes a new tab, if successful return the tab, returns None otherwise

show_plus_button_context_menu(*global_pos*)

Opens a context menu for the toolbar.

Parameters
global_pos (*QPoint*) – menu position on screen

make_context_menu(*index*)

Creates a context menu for given tab.

Parameters
index (*int*) – tab index

Returns
context menu or None if tab was not found

Return type
QMenu

_insert_statusbar_button(*button*)

Inserts given button to the 'beginning' of the status bar and decorates it with a shooting label.

Parameters
button (OpenFileButton) –

insert_open_file_button(*file_path*, *progress*, *is_sqlite*)

_open_sqlite_url(*url*, *codename*)

Opens sqlite url.

show_user_guide(*checked=False*)

Opens Spine db editor documentation page in browser.

class spinetoolbox.spine_db_editor.widgets.multi_spine_db_editor.**_CustomStatusBar**(*parent=None*)

Bases: PySide6.QtWidgets.QStatusBar

spinetoolbox.spine_db_editor.widgets.pivot_table_header_view

Contains custom QHeaderView for the pivot table.

Module Contents

Classes

PivotTableHeaderView	Header view for the pivot table.
ParameterValuePivotHeaderView	Header view for the pivot table in parameter value and index expansion mode.
ScenarioAlternativePivotHeaderView	Header view for the pivot table in parameter value and index expansion mode.

class spinetoolbox.spine_db_editor.widgets.pivot_table_header_view.**PivotTableHeaderView**(*orientation*, *area*, *pivot_table_view*)

Bases: PySide6.QtWidgets.QHeaderView

Header view for the pivot table.

Parameters

- **orientation** (*Qt.Orientation*) – Qt.Orientation.Horizontal or Qt.Orientation.Vertical
- **area** (*str*) – which pivot area the header represents: "columns", "rows" or "frozen"
- **pivot_table_view** (PivotTableView) – parent view

property area

header_dropped

dragEnterEvent(*event*)

dragMoveEvent(*event*)

dropEvent(*event*)

class spinetoolbox.spine_db_editor.widgets.pivot_table_header_view.**ParameterValuePivotHeaderView**(*orientat*, *area*, *pivot_ta*

Bases: *PivotTableHeaderView*

Header view for the pivot table in parameter value and index expansion mode.

Parameters

- **orientation** (*Qt.Orientation*) – Qt.Orientation.Horizontal or Qt.Orientation.Vertical
- **area** (*str*) – which pivot area the header represents: "columns", "rows" or "frozen"
- **pivot_table_view** (PivotTableView) – parent view

_column_selection()

Lists current column's indexes that contain some data.

Returns
column indexes

Return type
list of QModelIndex

_add_column_to_plot(*action*)

Adds a single column to existing plot window.

_plot_column()

Plots a single column not the selection.

_column_indexes(*column*)

Makes indexes for given column.

Parameters
column (*int*) – column

Returns
column indexes

Return type
list of QModelIndex

_set_x_flag()

Sets the X flag for a column.

contextMenuEvent(*event*)

Shows context menu.

Parameters
event (*QContextMenuEvent*) –

class spinetoolbox.spine_db_editor.widgets.pivot_table_header_view.**ScenarioAlternativePivotHeaderView**(*o*
a
p

Bases: *PivotTableHeaderView*

Header view for the pivot table in parameter value and index expansion mode.

Parameters

- **orientation** (*Qt.Orientation*) – Qt.Orientation.Horizontal or Qt.Orientation.Vertical
- **area** (*str*) – which pivot area the header represents: "columns", "rows" or "frozen"
- **pivot_table_view** (PivotTableView) – parent view

context_menu_requested

Requests a header context menu be shown at given global position.

contextMenuEvent(*event*)

spinetoolbox.spine_db_editor.widgets.scenario_generator

Contains a dialog for generating scenarios from selected alternatives.

Module Contents

Classes

_ScenarioNameResolution	Generic enumeration.
ScenarioGenerator	A dialog where users can generate scenarios from given alternatives.

Functions

_ensure_unique(scenario_alternatives)	Removes duplicate scenario alternatives.
_find_base_alternative(names)	Returns the name of a 'base' alternative or empty string if not found.
_suffix(item_count)	Returns a formattable string with enough zero padding to hold item_count digits.

class spinetoolbox.spine_db_editor.widgets.scenario_generator._ScenarioNameResolution

Bases: enum.Enum

Generic enumeration.

Derive from this class to define new enumerations.

NO_CONFLICT

OVERWRITE

LEAVE_AS_IS

CANCEL_OPERATION

class spinetoolbox.spine_db_editor.widgets.scenario_generator.ScenarioGenerator(*parent*, *db_map*, *alternatives*, *spine_db_editor*)

Bases: PySide6.QtWidgets.QWidget

A dialog where users can generate scenarios from given alternatives.

Parameters

- **parent** (*QWidget*) – parent widget
- **db_map** (*DiffDatabaseMapping*) – database mapping that contains the alternatives
- **alternatives** (*Iterable of CacheItem*) – alternatives from which the scenarios are generated
- **spine_db_editor** (SpineDBEditor) – database editor instance

_TYPE_LABELS = ('All combinations', 'Scenario for each alternative')

accept()

Generates scenarios and closes the dialog.

The operation may get cancelled by user if there are conflicts in scenario names.

_generate_scenarios(*new_scenarios*, *scenarios_to_modify*, *scenario_alternatives*)

Generates scenarios with all possible combinations of given alternatives.

Parameters

- **new_scenarios** (*Iterable of str*) – names of new scenarios to create
- **scenarios_to_modify** (*Iterable of str*) – names of scenarios to modify
- **scenario_alternatives** (*list of list*) – alternative items for each scenario

_check_existing_scenarios(*proposed_scenario_names*, *existing_scenario_names*)

Checks if proposed scenarios exist, and if so, prompts users what to do.

Parameters

- **proposed_scenario_names** (*Iterable of str*) – proposed scenario names
- **existing_scenario_names** (*set of str*) – existing scenario names

Returns

action to take

Return type
_ScenarioNameResolution

_enable_base_alternative(*check_box_state*)

Enables and disables base alternative combo box.

Parameters
check_box_state (*int*) – state of 'Use base alternative' check box

_insert_base_alternative(*scenario_alternatives*)

Prepends base alternative to scenario alternatives if it has been enabled.

If base alternative is already in scenario alternatives, make sure it comes first.

Parameters
scenario_alternatives (*list of list*) – scenario alternatives

spinetoolbox.spine_db_editor.widgets.scenario_generator.**_ensure_unique**(*scenario_alternatives*)

Removes duplicate scenario alternatives.

Parameters
scenario_alternatives (*list of list*) – scenario alternatives

spinetoolbox.spine_db_editor.widgets.scenario_generator.**_find_base_alternative**(*names*)

Returns the name of a 'base' alternative or empty string if not found.

Basically, checks if "Base" is in names, otherwise searches for the first case-insensitive version of "base".

Parameters
names (*list of str*) – alternative names

Returns
base alternative name

Return type
str

spinetoolbox.spine_db_editor.widgets.scenario_generator.**_suffix**(*item_count*)

Returns a formattable string with enough zero padding to hold item_count digits.

Parameters
item_count (*int*) – maximum number of items

Returns
string in the form '{:0n}' where n is the number of digits in item_count

Return type
str

spinetoolbox.spine_db_editor.widgets.select_graph_parameters_dialog

Classes for custom QDialogs to add items to databases.

Module Contents

Classes

SelectGraphParametersDialog	
ParameterNameDelegate	A delegate for the database name.

class spinetoolbox.spine_db_editor.widgets.select_graph_parameters_dialog.**SelectGraphParametersDialog**(*pa*
pa
ra
e-
te

Bases: PySide6.QtWidgets.QDialog

selection_made

resize_columns()

accept()

class spinetoolbox.spine_db_editor.widgets.select_graph_parameters_dialog.**ParameterNameDelegate**(*parent*,
db_mngr,
**db_map*

Bases: PySide6.QtWidgets.QStyledItemDelegate

A delegate for the database name.

setModelData(*editor*, *model*, *index*)

Send signal.

setEditorData(*editor*, *index*)

Do nothing. We're setting editor data right away in createEditor.

updateEditorGeometry(*editor*, *option*, *index*)

_close_editor(*editor*, *index*)

Closes editor. Needed by SearchBarEditor.

createEditor(*parent*, *option*, *index*)

Returns editor.

spinetoolbox.spine_db_editor.widgets.spine_db_editor

Contains the SpineDBEditor class.

Module Contents

Classes

SpineDBEditorBase	Base class for SpineDBEditor (i.e. Spine database editor).
SpineDBEditor	A widget to visualize Spine dbs.

class spinetoolbox.spine_db_editor.widgets.spine_db_editor.**SpineDBEditorBase**(*db_mngr*)

Bases: PySide6.QtWidgets.QMainWindow

Base class for SpineDBEditor (i.e. Spine database editor).

Parameters
db_mngr (SpineDBManager) – The manager to use

property toolbox

property settings_subgroup

property db_names

property first_db_map

property db_url_codenames

msg

msg_error

file_exported

filepath, progress between 0 and 1, True if sqlite file

static is_db_map_editor()

Always returns True as SpineDBEditors are truly database editors.

Unless, of course, the database can one day be opened in read-only mode. In that case this method should return False.

Returns
Always True

Return type
bool

load_db_urls(*db_url_codenames*, *create=False*, *update_history=True*, *window=False*)

init_add_undo_redo_actions()

load_previous_urls(*_=False*)

load_next_urls(*_=False*)

open_db_file(*_=False*)

add_db_file(*_=False*)

create_db_file(*_=False*)

reset_docs()

Resets the layout of the dock widgets for this URL

_make_docks_menu()

Returns a menu with all dock toggle/view actions. Called by `self.add_main_menu()`.

Returns
QMenu

add_main_menu()

Adds a menu with main actions to toolbar.

_browse_commits()

connect_signals()

Connects signals to slots.

vacuum(*_checked=False*)

update_undo_redo_actions(*_*)

_replace_undo_redo_actions(*new_undo_action*, *new_redo_action*)

_refresh_undo_redo_actions()

update_commit_enabled(*_clean=False*)

init_models()

Initializes models.

add_message(*msg*)

Pushes message to notification stack.

Parameters
msg (*str*) – String to show in the notification

refresh_copy_paste_actions()

Runs when menus are about to show. Enables or disables actions according to selection status.

copy(*checked=False*)

Copies data to clipboard.

paste(*checked=False*)

Pastes data from clipboard.

import_data(*data*)

Imports data to all database mappings open in the editor.

Parameters
data (*dict*) – data to import

import_file(*checked=False*)

Imports file.

It supports SQLite, JSON, and Excel.

import_from_json(*file_path*)

import_from_sqlite(*file_path*)

import_from_excel(*file_path*)

show_mass_export_items_dialog(*checked=False*)

Shows dialog for user to select dbs and items for export.

_store_export_settings(*state*)

Stores export items dialog settings.

_clean_up_export_items_dialog()

Cleans up export items dialog.

export_session(*checked=False*)

Exports changes made in the current session.

mass_export_items(*db_map_item_types*)

duplicate_entity(*entity_item*)

Duplicates an entity.

Parameters
entity_item (EntityItem) –

duplicate_scenario(*db_map*, *scen_id*)

Duplicates a scenario.

Parameters

- **db_map** (*DiffDatabaseMapping*) –
- **scen_id** (*int*) –

export_data(*db_map_ids_for_export*)

Exports data from given dictionary into a file.

Parameters
db_map_ids_for_export – Dictionary mapping db maps to keyword arguments for spinedb_api.export_data

refresh_session(*checked=False*)

commit_session(*checked=False*)

Commits dirty database maps.

rollback_session(*checked=False*)

Rolls back dirty database maps.

receive_session_committed(*db_maps*, *cookie*)

receive_session_rolled_back(*db_maps*)

receive_session_refreshed(*db_maps*)

show_mass_remove_items_form(*checked=False*)

Opens the purge items dialog.

_store_purge_settings(*state*)

Stores Purge items dialog state.

Parameters
state (*dict*) – dialog state

_clean_up_purge_items_dialog()

Removes references to purge items dialog.

show_parameter_value_editor(*index*, *plain=False*)

Shows the parameter_value editor for the given index of given table view.

receive_error_msg(*db_map_error_log*)

_update_export_enabled()

Updates export enabled.

_log_items_change(*msg*)

Enables or disables actions and informs the user about what just happened.

_handle_items_added(*item_type*, *db_map_data*)

_handle_items_updated(*item_type*, *db_map_data*)

_handle_items_removed(*item_type*, *db_map_data*)

restore_ui(*view_type*, *fresh=False*)

Restores UI state from previous session.

Parameters

- **view_type** (*str*) – What the selected view type is.
- **fresh** (*bool*) – If true, the view specified with subgroup will be applied, instead of loading the previous window state of the said view.

save_window_state()

Saves window state parameters (size, position, state) via QSettings.

tear_down()

Performs clean up duties.

Returns
True if editor is ready to close, False otherwise

Return type
bool

_prompt_to_commit_changes()

Prompts the user to commit or rollback changes to 'dirty' db maps.

Returns
QMessageBox status code

Return type
int

_get_commit_msg(*db_names*)

Prompts user for commit message.

Parameters
db_names (*Iterable of str*) – database names

Returns
commit message

Return type
str

_get_rollback_confirmation(*db_names*)

Prompts user for confirmation before rolling back the session.

Parameters
db_names (*Iterable of str*) – database names

Returns
True if user confirmed, False otherwise

Return type
bool

_purge_change_notifiers()

Tears down change notifiers.

closeEvent(*event*)

Handle close window.

Parameters
event (*QCloseEvent*) – Closing event

static _get_base_dir()

class spinetoolbox.spine_db_editor.widgets.spine_db_editor.**SpineDBEditor**(*db_mngr*, *db_url_codenames=None*)

Bases: *spinetoolbox.spine_db_editor.widgets.tabular_view_mixin.TabularViewMixin*, *spinetoolbox.spine_db_editor.widgets.graph_view_mixin.GraphViewMixin*, *spinetoolbox.spine_db_editor.widgets.stacked_view_mixin.StackedViewMixin*, *spinetoolbox.spine_db_editor.widgets.tree_view_mixin.TreeViewMixin*, *SpineDBEditorBase*

A widget to visualize Spine dbs.

Initializes everything.

Parameters
db_mngr (SpineDBManager) – The manager to use

pinned_values_updated

emit_pinned_values_updated()

connect_signals()

Connects signals to slots.

init_models()

Initializes models.

_restart_timer_refresh_tab_order(*_visible=False*)

_refresh_tab_order()

tabify_and_raise(*docks*)

Tabifies docks in given list, then raises the first.

Parameters
docks (*list*) –

restore_dock_widgets()

Docks all floating and or hidden QDockWidgets back to the window.

update_last_view()

begin_style_change()

Begins a style change operation.

end_style_change()

Ends a style change operation.

apply_stacked_style(*_checked=None*)

Applies the stacked style, inspired in the former tree view.

_finish_stacked_style()

apply_pivot_style(*_checked=None*)

Applies the pivot style, inspired in the former tabular view.

apply_graph_style(*_checked=None*)

Applies the graph style, inspired in the former graph view.

static _get_base_dir()

spinetoolbox.spine_db_editor.widgets.stacked_view_mixin

Contains the StackedViewMixin class.

Module Contents

Classes

StackedViewMixin	Provides stacked parameter tables for the Spine db editor.

class spinetoolbox.spine_db_editor.widgets.stacked_view_mixin.**StackedViewMixin**(**args*, ***kwargs*)

Provides stacked parameter tables for the Spine db editor.

connect_signals()

Connects signals to slots.

init_models()

Initializes models.

show_element_name_list_editor(*index*, *entity_class_id*, *db_map*)

Shows the element name list editor.

Parameters

- **index** (*QModelIndex*) –
- **entity_class_id** (*int*) –
- **db_map** (*DatabaseMapping*) –

_set_default_parameter_data(*index=None*)

Sets default rows for parameter models according to given index.

Parameters
index (*QModelIndex*) – an index of the entity tree

set_default_parameter_data(*default_data*, *default_db_map*)

clear_all_filters()

_reset_filters()

Resets filters.

_handle_graph_selection_changed(*selected_items*)

Resets filter according to graph selection.

_handle_entity_tree_selection_changed_in_parameter_tables(*selected_indexes*)

Resets filter according to entity tree selection.

_handle_alternative_selection_changed(*selected_db_map_alt_ids*)

Resets filter according to selection in alternative tree view.

_handle_scenario_alternative_selection_changed(*selected_db_map_alt_ids*)

Resets filter according to selection in scenario tree view.

_update_alternative_selection(*selected_db_map_alt_ids*, *other_tree_view*, *this_tree_view*)

Combines alternative selections from alternative and scenario tree views.

Parameters

- **selected_db_map_alt_ids** (*dict*) – mapping from database map to set of alternative ids
- **other_tree_view** (AlternativeTreeView *or* ScenarioTreeView) – tree view whose selection didn't change
- **this_tree_view** (AlternativeTreeView *or* ScenarioTreeView) – tree view whose selection changed

_clear_all_other_selections(*current*, *other=None*)

Clears all the other selections besides the one that was just made.

Parameters

- **current** – the tree where the selection that was just made
- **other** (*optional*) – other optional tree

tear_down()

spinetoolbox.spine_db_editor.widgets.tabular_view_header_widget

Contains TabularViewHeaderWidget class.

Module Contents

Classes

TabularViewHeaderWidget	A draggable QWidget.

class spinetoolbox.spine_db_editor.widgets.tabular_view_header_widget.**TabularViewHeaderWidget**(*identifier*, *area*, *menu=None*, *parent=None*)

Bases: PySide6.QtWidgets.QFrame

A draggable QWidget.

Parameters

- **identifier** (*str*) –
- **area** (*str*) – either “rows”, “columns”, or “frozen”
- **menu** (*FilterMenu, optional*) –
- **parent** (*QWidget, optional*) – Parent widget

property identifier

property area

header_dropped

_H_MARGIN = 3

_SPACING = 16

mousePressEvent(*event*)

Register drag start position

mouseMoveEvent(*event*)

Start dragging action if needed

mouseReleaseEvent(*event*)

Forget drag start position

dragEnterEvent(*event*)

dropEvent(*event*)

spinetoolbox.spine_db_editor.widgets.tabular_view_mixin

Contains TabularViewMixin class.

Module Contents

Classes

TabularViewMixin	Provides the pivot table and its frozen table for the Database editor.

class spinetoolbox.spine_db_editor.widgets.tabular_view_mixin.**TabularViewMixin**(**args*, ***kwargs*)

Provides the pivot table and its frozen table for the Database editor.

property current_dimension_id_list

property first_current_entity_class

property current_dimension_name_list

property current_dimension_ids

_PARAMETER_VALUE = '&Value'

_INDEX_EXPANSION = '&Index'

_ELEMENT = 'E&lement'

_SCENARIO_ALTERNATIVE = '&Scenario'

_PARAMETER = 'parameter'

_ALTERNATIVE = 'alternative'

_INDEX = 'index'

populate_pivot_action_group()

connect_signals()

Connects signals to slots.

refresh_views()

update_filter_menus(*action*)

_needs_to_update_headers(*item_type*, *db_map_data*)

_reload_pivot_table_if_needed(*item_type*, *db_map_data*)

_connect_pivot_table_header_signals()

Connects signals of pivot table's header views.

init_models()

Initializes models.

_set_model_data(*index*, *value*)

static _is_class_index(*index*)

Returns whether the given tree index is a class index.

Parameters
index (*QModelIndex*) – index from object or relationship tree

Returns
bool

_handle_pivot_action_triggered(*action*)

_handle_pivot_table_visibility_changed(*visible*)

_handle_entity_tree_selection_changed_in_pivot_table(*selected_indexes*)

_update_class_attributes(*current_index*)

Updates current class id and name.

static _get_current_class_item(*current_index*)

get_pivot_preferences()

Returns saved pivot preferences.

Returns
pivot tuple, or None if no preference stored

Return type
tuple, NoneType

do_reload_pivot_table()

Reloads pivot table.

_can_build_pivot_table()

clear_pivot_table()

wipe_out_headers()

make_pivot_headers()

Turns top left indexes in the pivot table into TabularViewHeaderWidget.

_resize_pivot_header_columns()

_make_inserted_frozen_headers(*parent_index*, *first_column*, *last_column*)

Turns the first row of columns in the frozen table into TabularViewHeaderWidgets.

Parameters

- **parent_index** (*QModelIndex*) – frozen table column's parent index
- **first_column** (*int*) – first inserted column
- **last_column** (*int*) – last inserted column

_make_all_frozen_headers()

Turns the first row of columns in the frozen table into TabularViewHeaderWidgets.

_make_frozen_headers(*first_column*, *last_column*)

_check_frozen_value_selected(*parent*, *first_row*, *last_row*)

Ensures that at least one row is selected in frozen table when number of rows change.

create_filter_menu(*identifier*)

Returns a filter menu for given filterable item.

Parameters
identifier (*str*) – item identifier

Returns
filter menu corresponding to identifier

Return type
TabularViewDBItemFilterMenu

create_header_widget(*identifier*, *area*, *with_menu=True*)

Returns a TabularViewHeaderWidget for given object_class identifier.

Parameters

- **identifier** (*str*) –
- **area** (*str*) –
- **with_menu** (*bool*) –

Returns
TabularViewHeaderWidget

static _get_insert_index(*pivot_list*, *catcher*, *position*)

Returns an index for inserting a new element in the given pivot list.

Returns
int

handle_header_dropped(*dropped*, *catcher*, *position=''*)

Updates pivots when a header is dropped.

Parameters

- **dropped** (TabularViewHeaderWidget) – drag source widget
- **catcher** (TabularViewHeaderWidget *or* PivotTableHeaderView *or* FrozenTableView) – drop target widget
- **position** (*str*) – either “before”, “after”, or “”

_change_selected_frozen_row(*current*, *previous*)

Sets the frozen value from selection in frozen table.

_update_current_index_if_need()

Ensures selected frozen row corresponds to current index.

Frozen table gets sorted from time to time possibly changing the selected row.

change_filter(*identifier*, *valid_values*, *has_filter*)

_add_values_to_frozen_table(*frozen_values*)

Adds values to frozen table.

Parameters
frozen_values (*set of tuple*) – values to add

_remove_values_from_frozen_table(*frozen_values*)

Removes values from frozen table.

Parameters
frozen_values (*set of tuple*) – values to remove

reload_frozen_table()

Resets the frozen model according to new selection in entity trees.

find_frozen_values(*frozen*)

Returns a list of tuples containing unique values for the frozen indexes.

Parameters
frozen (*tuple*) – A tuple of currently frozen indexes

Returns
frozen value

Return type
list

_change_frozen_value()

Updated frozen value according to selected row in Frozen table.

receive_session_rolled_back(*db_maps*)

Reacts to session rolled back event.

accepts_entity_class_item(*item*, *db_map*)

accepts_entity_item(*item*, *db_map*)

accepts_parameter_item(*item*, *db_map*)

accepts_element_item(*item*, *db_map*)

accepts_ith_element_item(*i*, *item*, *db_map*)

_frozen_table_reload_disabled()

closeEvent(*event*)

spinetoolbox.spine_db_editor.widgets.tree_view_mixin

Contains the TreeViewMixin class.

Module Contents

Classes

TreeViewMixin	Provides object and relationship trees for the Spine db editor.

class spinetoolbox.spine_db_editor.widgets.tree_view_mixin.**TreeViewMixin**(**args*, ***kwargs*)

Provides object and relationship trees for the Spine db editor.

init_models()
Initializes models.

_db_map_ids(*indexes*)

export_selected(*selected_indexes*)
Exports data from given indexes in the entity tree.

show_add_entity_classes_form(*parent_item*)
Shows dialog to add new entity classes.

show_add_entities_form(*parent_item*)
Shows dialog to add new entities.

show_add_entity_group_form(*entity_class_item*)
Shows dialog to add new entity group.

show_manage_members_form(*entity_item*)
Shows dialog to manage an entity group.

show_manage_elements_form(*parent_item*)

show_select_superclass_form(*entity_class_item*)

edit_entity_tree_items(*selected_indexes*)
Starts editing given indexes.

show_edit_entity_classes_form(*items*)

show_edit_entities_form(*items*)

remove_entity_tree_items(*selected_indexes*)
Shows form to remove items from object treeview.

show_remove_entity_tree_items_form(*selected*)

spinetoolbox.spine_db_editor.widgets.url_toolbar

Contains the UrlToolBar class and helpers.

Module Contents

Classes

UrlToolBar

_FilterWidget

_FilterArrayWidget

_DBListWidget

_UrlFilterDialog

class spinetoolbox.spine_db_editor.widgets.url_toolbar.**UrlToolBar**(*db_editor*)

Bases: PySide6.QtWidgets.QToolBar

property line_edit

_add_open_project_url_menu()

_update_open_project_url_menu()

_open_ds_url(*action*)

add_main_menu(*menu*)

_update_history_actions_availability()

add_urls_to_history(*db_urls*)

Adds url to history.

Parameters
db_urls (*list of str*) –

get_previous_urls()

Returns previous urls in history.

Returns
list of str

get_next_urls()

Returns next urls in history.

Returns
list of str

_handle_line_edit_return_pressed()

set_current_urls(*urls*)

_show_filter_menu(*_checked=False*)

class spinetoolbox.spine_db_editor.widgets.url_toolbar.**_FilterWidget**(*db_mngr*, *db_map*, *item_type*, *filter_type*, *active_item*, *parent=None*)

Bases: PySide6.QtWidgets.QTreeWidget

sizeHint()

filter_config()

class spinetoolbox.spine_db_editor.widgets.url_toolbar.**_FilterArrayWidget**(*db_mngr*, *db_map*, *parent=None*)

Bases: PySide6.QtWidgets.QWidget

filter_selection_changed

filtered_url_codename()

sizeHint()

moveEvent(*ev*)

class spinetoolbox.spine_db_editor.widgets.url_toolbar._DBListWidget(*db_mngr*, *db_maps*, *parent=None*)

Bases: PySide6.QtWidgets.QTreeWidget

db_filter_selection_changed

sizeHint()

filtered_url_codenames()

class spinetoolbox.spine_db_editor.widgets.url_toolbar._UrlFilterDialog(*db_mngr*, *db_maps*, *parent=None*)

Bases: PySide6.QtWidgets.QDialog

filter_accepted

sizeHint()

_update_filter_enabled()

accept()

Submodules

spinetoolbox.spine_db_editor.graphics_items

Classes for drawing graphics items on graph view's QGraphicsScene.

Module Contents

Classes

EntityItem	**param spine_db_editor** 'owner'
ArcItem	Connects two EntityItems.
CrossHairsItem	Creates new relationships directly in the graph.
CrossHairsEntityItem	Represents the relationship that's being created using the CrossHairsItem.
CrossHairsArcItem	Connects a CrossHairsEntityItem with the CrossHairsItem,
EntityLabelItem	Provides a label for EntityItem.
BgItem	
_ResizableQGraphicsSvgItem	
_Resizer	

class spinetoolbox.spine_db_editor.graphics_items.**EntityItem**(*spine_db_editor*, *x*, *y*, *extent*, *db_map_ids*, *offset=None*)

Bases: PySide6.QtWidgets.QGraphicsRectItem

Parameters

- **spine_db_editor** (SpineDBEditor) – 'owner'
- **x** (*float*) – x-coordinate of central point
- **y** (*float*) – y-coordinate of central point
- **extent** (*int*) – Preferred extent
- **db_map_ids** (*tuple*) – tuple of (db_map, id) tuples

property **has_dimensions**

property **db_map_ids**

property **original_db_map_ids**

property **name**

property **first_entity_class_id**

property **entity_class_name**

property **dimension_id_list**

property **dimension_name_list**

property **byname**

property **element_name_list**

property **element_byname_list**

property **first_db_map_id**

property **first_id**

property **first_db_map**

property **display_data**

property **display_database**

property **db_maps**

clone()

element_id_list(*db_map*)

entity_class_id(*db_map*)

entity_class_ids(*db_map*)

entity_id(*db_map*)

db_map_data(*db_map*)

db_map_id(*db_map*)

db_items(*db_map*)

boundingRect()

set_pos(*x*, *y*)

move_by(*dx*, *dy*)

_snap(*x*, *y*)

has_unique_key()

Returns whether or not the item still has a single key in all the databases it represents.

Returns
bool

_get_name()

_get_prop(*getter*, *index*)

_get_color(*index=None*)

_get_arc_width(*index=None*)

_get_vertex_radius(*index=None*)

_has_name()

set_up()

update_props(*index*)

_update_bg()

refresh_icon()

Refreshes the icon.

_update_renderer(*color*, *resize=True*)

_install_renderer(*resize=True*)

_make_tool_tip()

default_parameter_data()

Return data to put as default in a parameter table when this item is selected.

shape()

Returns a shape containing the entire bounding rect, to work better with icon transparency.

set_highlight_color(*color*)

paint(*painter*, *option*, *widget=None*)

Shows or hides the selection halo.

_paint_as_selected()

_paint_as_deselected()

add_arc_item(*arc_item*)

Adds an item to the list of arcs.

Parameters
arc_item (ArcItem) –

update_entity_pos()

do_update_entity_pos()

apply_zoom(*factor*)

Applies zoom.

Parameters
factor (*float*) – The zoom factor.

apply_rotation(*angle*, *center*)

Applies rotation.

Parameters

- **angle** (*float*) – The angle in degrees.
- **center** (*QPointF*) – Rotates around this point.

mouseMoveEvent(*event*)

Moves the item and all connected arcs.

Parameters
event (*QGraphicsSceneMouseEvent*) –

update_arcs_line()

Moves arc items.

_update_arcs(*color*, *arc_width*)

_update_circle(*color*, *vertex_radius*)

itemChange(*change*, *value*)

Keeps track of item's movements on the scene. Rotates svg item if the relationship is 2D. This makes it possible to define e.g. an arow icon for relationships that express direction.

Parameters

- **change** (*GraphicsItemChange*) – a flag signalling the type of the change
- **value** – a value related to the change

Returns
the same value given as input

setVisible(*on*)

Sets visibility status for this item and all arc items.

Parameters
on (*bool*) –

_make_menu()

contextMenuEvent(*e*)

Shows context menu.

Parameters
e (*QGraphicsSceneMouseEvent*) – Mouse event

remove_db_map_ids(*db_map_ids*)

Removes db_map_ids.

add_db_map_ids(*db_map_ids*)

_rotate_svg_item()

mouseDoubleClickEvent(*e*)

_duplicate()

_refresh_db_map_entity_class_lists()

_populate_expand_collapse_menu(*menu*)

Populates the 'Expand' or 'Collapse' menu.

Parameters
menu (*QMenu*) –

_populate_connect_entities_menu(*menu*)

Populates the 'Add relationships' menu.

Parameters
menu (*QMenu*) –

_get_db_map_entity_ids_to_expand_or_collapse(*action*)

_expand(*action*)

_collapse(*action*)

_start_connecting_entities(*action*)

class spinetoolbox.spine_db_editor.graphics_items.**ArcItem**(*ent_item*, *el_item*, *width*)

Bases: PySide6.QtWidgets.QGraphicsPathItem

Connects two EntityItems.

Parameters

- **ent_item** (*spinetoolbox.widgets.graph_view_graphics_items.EntityItem*) – entity item
- **el_item** (*spinetoolbox.widgets.graph_view_graphics_items.EntityItem*) – element item
- **width** (*float*) – Preferred line width

clone(*entity_items*)

_make_pen()

moveBy(*dx*, *dy*)

Does nothing. This item is not moved the regular way, but follows the EntityItems it connects.

update_line()

update_color(*color*)

apply_value(*factor*, *sign*)

mousePressEvent(*event*)

Accepts the event so it's not propagated.

other_item(*item*)

apply_zoom(*factor*)

Applies zoom.

Parameters

factor (*float*) – The zoom factor.

_update_width()

_move_gradient(*factor*, *sign*)

_do_move_gradient()

class spinetoolbox.spine_db_editor.graphics_items.**CrossHairsItem**(**args*, ***kwargs*)

Bases: *EntityItem*

Creates new relationships directly in the graph.

Parameters

- **spine_db_editor** (SpineDBEditor) – 'owner'
- **x** (*float*) – x-coordinate of central point
- **y** (*float*) – y-coordinate of central point
- **extent** (*int*) – Preferred extent
- **db_map_ids** (*tuple*) – tuple of (db_map, id) tuples

property entity_class_name

property name

property has_dimensions

_make_tool_tip()

_has_name()

refresh_icon()

Refreshes the icon.

set_plus_icon()

set_check_icon()

set_normal_icon()

set_ban_icon()

set_icon(*unicode*, *color=0*)

Refreshes the icon.

_snap(*x*, *y*)

mouseMoveEvent(*event*)

Moves the item and all connected arcs.

Parameters
event (*QGraphicsSceneMouseEvent*) –

contextMenuEvent(*e*)

Shows context menu.

Parameters
e (*QGraphicsSceneMouseEvent*) – Mouse event

class spinetoolbox.spine_db_editor.graphics_items.**CrossHairsEntityItem**(**args*, ***kwargs*)

Bases: *EntityItem*

Represents the relationship that's being created using the CrossHairsItem.

Parameters

- **spine_db_editor** (SpineDBEditor) – 'owner'
- **x** (*float*) – x-coordinate of central point
- **y** (*float*) – y-coordinate of central point
- **extent** (*int*) – Preferred extent
- **db_map_ids** (*tuple*) – tuple of (db_map, id) tuples

property has_dimensions

_make_tool_tip()

_has_name()

refresh_icon()

Refreshes the icon.

contextMenuEvent(*e*)

Shows context menu.

Parameters
e (*QGraphicsSceneMouseEvent*) – Mouse event

class spinetoolbox.spine_db_editor.graphics_items.**CrossHairsArcItem**(*ent_item*, *el_item*, *width*)

Bases: *ArcItem*

Connects a CrossHairsEntityItem with the CrossHairsItem, and with all the EntityItem's in the relationship so far.

Parameters

- **ent_item** (*spinetoolbox.widgets.graph_view_graphics_items.EntityItem*) – entity item
- **el_item** (*spinetoolbox.widgets.graph_view_graphics_items.EntityItem*) – element item
- **width** (*float*) – Preferred line width

_make_pen()

class spinetoolbox.spine_db_editor.graphics_items.**EntityLabelItem**(*entity_item*)

Bases: PySide6.QtWidgets.QGraphicsTextItem

Provides a label for EntityItem.

Initializes item.

Parameters
entity_item (*spinetoolbox.widgets.graph_view_graphics_items.EntityItem*) – The parent item.

entity_name_edited

boundingRect()

setPlainText(*text*)

Set texts and resets position.

Parameters
text (*str*) –

reset_position()

Adapts item geometry so text is always centered.

class spinetoolbox.spine_db_editor.graphics_items.**BgItem**(*svg*, *parent=None*)

Bases: PySide6.QtWidgets.QGraphicsRectItem

class **Anchor**

Bases: enum.Enum

Generic enumeration.

Derive from this class to define new enumerations.

TL

TR

BL

BR

_getter_setter

_cursors

clone()

hoverEnterEvent(*ev*)

hoverLeaveEvent(*ev*)

apply_zoom(*factor*)

_place_resizers()

_resize(*anchor*, *delta*, *strong*)

_do_resize(*rect*, *strong*)

fit_rect(*rect*)

scene_rect()

class spinetoolbox.spine_db_editor.graphics_items.**_ResizableQGraphicsSvgItem**(**args*, ***kwargs*)

Bases: PySide6.QtSvgWidgets.QGraphicsSvgItem

resize(*width*, *height*)

setSharedRenderer(*renderer*)

boundingRect()

paint(*painter*, *options*, *widget*)

class spinetoolbox.spine_db_editor.graphics_items.**_Resizer**(*rect=QRectF(0, 0, 20, 20)*, *parent=None*)

Bases: PySide6.QtWidgets.QGraphicsRectItem

class SignalsProvider

Bases: PySide6.QtCore.QObject

resized

mousePressEvent(*ev*)

mouseMoveEvent(*ev*)

mouseReleaseEvent(*ev*)

spinetoolbox.spine_db_editor.helpers

Helpers and utilities for Spine Database editor.

Module Contents

Functions

string_to_display_icon(x)	Converts a 'foreign' string (from e.g. Excel) to entity class display icon.
string_to_bool(x)	Converts a 'foreign' string (from e.g. Excel) to boolean.

Attributes

TRUE_STRING
FALSE_STRING
GENERIC_TRUE
GENERIC_FALSE

spinetoolbox.spine_db_editor.helpers.**string_to_display_icon**(*x*)

Converts a 'foreign' string (from e.g. Excel) to entity class display icon.

Parameters
x (*str*) – string to convert

Returns
display icon or None if conversion failed

Return type
int

spinetoolbox.spine_db_editor.helpers.**TRUE_STRING = 'true'**

spinetoolbox.spine_db_editor.helpers.**FALSE_STRING = 'false'**

spinetoolbox.spine_db_editor.helpers.**GENERIC_TRUE**

spinetoolbox.spine_db_editor.helpers.**GENERIC_FALSE**

spinetoolbox.spine_db_editor.helpers.**string_to_bool**(*x*)

Converts a 'foreign' string (from e.g. Excel) to boolean.

Parameters
x (*str*) – string to convert

Returns
boolean value

Return type
bool

spinetoolbox.spine_db_editor.main

Module Contents

Functions

main()	Launches Spine Db Editor as its own application.
_make_argument_parser()	Builds a command line argument parser.

spinetoolbox.spine_db_editor.main.**main**()

Launches Spine Db Editor as its own application.

spinetoolbox.spine_db_editor.main.**_make_argument_parser**()

Builds a command line argument parser.

Returns
parser

Return type
ArgumentParser

spinetoolbox.spine_db_editor.scenario_generation

Contains functions for automatically generating scenarios from a set of alternatives.

Module Contents

Functions

all_combinations(alternatives)	Creates all possible combinations of alternatives.
unique_alternatives(alternatives)	Creates all possible single-alternative scenarios.

spinetoolbox.spine_db_editor.scenario_generation.**all_combinations**(*alternatives*)

Creates all possible combinations of alternatives.

Parameters
alternatives (*Iterable of Any*) – alternatives

Returns
lists containing alternatives for each scenario

Return type
list of list

spinetoolbox.spine_db_editor.scenario_generation.**unique_alternatives**(*alternatives*)

Creates all possible single-alternative scenarios.

Parameters
alternatives (*Iterable of Any*) – alternatives

Returns
tuples containing alternatives for each scenario

Return type
list of list

spinetoolbox.widgets

Init file for widgets package. Intentionally empty.

Submodules

spinetoolbox.widgets.about_widget

A widget for presenting basic information about the application.

Module Contents

Classes

AboutWidget	About widget class.

class spinetoolbox.widgets.about_widget.**AboutWidget**(*toolbox*)

Bases: PySide6.QtWidgets.QWidget

About widget class.

Parameters
toolbox (ToolboxUI) – QMainWindow instance

copy_to_clipboard(_)

Copies package and Python info to clipboard.

calc_pos()

Calculate the top-left corner position of this widget in relation to main window position and size in order to show about window in the middle of the main window.

setup_license_text()

Add license to QTextBrowser.

keyPressEvent(*e*)

Close form when Escape, Enter, Return, or Space bar keys are pressed.

Parameters
e (*QKeyEvent*) – Received key press event.

closeEvent(*event=None*)

Handle close window.

Parameters
event (*QEvent*) – Closing event if 'X' is clicked.

mousePressEvent(*e*)

Save mouse position at the start of dragging.

Parameters
e (*QMouseEvent*) – Mouse event

mouseReleaseEvent(*e*)

Save mouse position at the end of dragging.

Parameters
e (*QMouseEvent*) – Mouse event

mouseMoveEvent(*e*)

Moves the window when mouse button is pressed and mouse cursor is moved.

Parameters
e (*QMouseEvent*) – Mouse event

spinetoolbox.widgets.add_project_item_widget

Widget shown to user when a new Project Item is created.

Module Contents

Classes

AddProjectItemWidget	A widget to query user's preferences for a new item.

class spinetoolbox.widgets.add_project_item_widget.**AddProjectItemWidget**(*toolbox*, *x*, *y*, *class_*, *spec=''*)

Bases: PySide6.QtWidgets.QWidget

A widget to query user's preferences for a new item.

toolbox

Parent widget

Type
ToolboxUI

x

X coordinate of new item

Type
int

y

Y coordinate of new item

Type
int

Initialize class.

connect_signals()

Connect signals to slots.

handle_name_changed()

Update label to show upcoming folder name.

handle_ok_clicked()

Check that given item name is valid and add it to project.

abstract call_add_item()

Creates new Item according to user's selections.

Must be reimplemented by subclasses.

keyPressEvent(*e*)

Close Setup form when escape key is pressed.

Parameters
e (*QKeyEvent*) – Received key press event.

closeEvent(*event=None*)

Handle close window.

Parameters

event (*QEvent*) – Closing event if 'X' is clicked.

spinetoolbox.widgets.add_up_spine_opt_wizard

Classes for custom QDialogs for julia setup.

Module Contents

Classes

_PageId	Enum where members are also (and must be) ints
AddUpSpineOptWizard	A wizard to install & upgrade SpineOpt.
IntroPage	
SelectJuliaPage	
CheckPreviousInstallPage	
AddUpSpineOptPage	A QWizards page with a log. Useful for pages that need to capture the output of a process.
SuccessPage	
FailurePage	
TroubleshootProblemsPage	
TroubleshootSolutionPage	
ResetRegistryPage	A QWizards page with a log. Useful for pages that need to capture the output of a process.
AddUpSpineOptAgainPage	A QWizards page with a log. Useful for pages that need to capture the output of a process.
TotalFailurePage	

Functions

_clear_layout(layout)	

class spinetoolbox.widgets.add_up_spine_opt_wizard.**_PageId**

Bases: enum.IntEnum

Enum where members are also (and must be) ints

Initialize self. See help(type(self)) for accurate signature.

INTRO

SELECT_JULIA

CHECK_PREVIOUS_INSTALL

ADD_UP_SPINE_OPT

SUCCESS

FAILURE

TROUBLESHOOT_PROBLEMS

TROUBLESHOOT_SOLUTION

RESET_REGISTRY

ADD_UP_SPINE_OPT_AGAIN

TOTAL_FAILURE

class spinetoolbox.widgets.add_up_spine_opt_wizard.**AddUpSpineOptWizard**(*parent*, *julia_exe*, *julia_project*)

Bases: PySide6.QtWidgets.QWizard

A wizard to install & upgrade SpineOpt.

Parameters

- **parent** (*QWidget*) – the parent widget (SettingsWidget)
- **julia_exe** (*str*) – path to Julia executable
- **julia_project** (*str*) – path to Julia project

class spinetoolbox.widgets.add_up_spine_opt_wizard.**IntroPage**(*parent*)

Bases: PySide6.QtWidgets.QWizardPage

nextId()

class spinetoolbox.widgets.add_up_spine_opt_wizard.**SelectJuliaPage**(*parent*, *julia_exe*, *julia_project*)

Bases: PySide6.QtWidgets.QWizardPage

initializePage()

_select_julia_exe(_)

_select_julia_project(_)

nextId()

class spinetoolbox.widgets.add_up_spine_opt_wizard.**CheckPreviousInstallPage**(*parent*)

Bases: PySide6.QtWidgets.QWizardPage

isComplete()

cleanupPage()

initializePage()

_handle_check_install_finished(*ret*)

nextId()

class spinetoolbox.widgets.add_up_spine_opt_wizard.**AddUpSpineOptPage**(*parent*)

Bases: *spinetoolbox.widgets.custom_qwidgets.QWizardProcessPage*

A QWizards page with a log. Useful for pages that need to capture the output of a process.

initializePage()

_handle_spine_opt_add_up_finished(*ret*)

nextId()

class spinetoolbox.widgets.add_up_spine_opt_wizard.**SuccessPage**(*parent*)

Bases: PySide6.QtWidgets.QWizardPage

initializePage()

nextId()

class spinetoolbox.widgets.add_up_spine_opt_wizard.**FailurePage**(*parent*)

Bases: PySide6.QtWidgets.QWizardPage

_handle_check_box_clicked(*checked=False*)

initializePage()

nextId()

class spinetoolbox.widgets.add_up_spine_opt_wizard.**TroubleshootProblemsPage**(*parent*)

Bases: PySide6.QtWidgets.QWizardPage

isComplete()

_show_log(*_=False*)

nextId()

class spinetoolbox.widgets.add_up_spine_opt_wizard.**TroubleshootSolutionPage**(*parent*)

Bases: PySide6.QtWidgets.QWizardPage

cleanupPage()

initializePage()

_initialize_page_solution1()

_initialize_page_solution2()

nextId()

class spinetoolbox.widgets.add_up_spine_opt_wizard.**ResetRegistryPage**(*parent*)

Bases: *spinetoolbox.widgets.custom_qwidgets.QWizardProcessPage*

A QWizards page with a log. Useful for pages that need to capture the output of a process.

initializePage()

_handle_registry_reset_finished(*ret*)

nextId()

class spinetoolbox.widgets.add_up_spine_opt_wizard.**AddUpSpineOptAgainPage**(*parent*)

Bases: *AddUpSpineOptPage*

A QWizards page with a log. Useful for pages that need to capture the output of a process.

nextId()

class spinetoolbox.widgets.add_up_spine_opt_wizard.**TotalFailurePage**(*parent*)

Bases: PySide6.QtWidgets.QWizardPage

nextId()

spinetoolbox.widgets.add_up_spine_opt_wizard.**_clear_layout**(*layout*)

spinetoolbox.widgets.array_editor

Contains an editor widget for array type parameter values.

Module Contents

Classes

ArrayEditor	Editor widget for Arrays.

class spinetoolbox.widgets.array_editor.**ArrayEditor**(*parent=None*)

Bases: PySide6.QtWidgets.QWidget

Editor widget for Arrays.

Parameters
parent (*QWidget, optional*) – parent widget

set_value(*value*)

Sets the parameter_value for editing in this widget.

Parameters
value (*Array*) – value for editing

value()

Returns the array currently being edited.

Returns
array

Return type
Array

_check_if_plotting_enabled(*type_name*)

Checks is array's data type allows the array to be plotted.

Parameters
type_name (*str*) – data type's name

_change_value_type(*type_name*)

open_value_editor(*index*)

Opens an editor widget for array element.

Parameters
index (*QModelIndex*) – element's index

_show_table_context_menu(*position*)

Shows the table's context menu.

Parameters
position (*QPoint*) – menu's position on the table

_update_plot(*topLeft=None*, *bottomRight=None*, *roles=None*)

Updates the plot widget.

_open_header_editor(*column*)

spinetoolbox.widgets.array_value_editor

An editor dialog for Array elements.

Module Contents

Classes

ArrayValueEditor	Editor widget for Array elements.

class spinetoolbox.widgets.array_value_editor.**ArrayValueEditor**(*index*, *value_type*, *parent=None*)

Bases: *spinetoolbox.widgets.parameter_value_editor_base.ParameterValueEditorBase*

Editor widget for Array elements.

Parameters

- **index** (*QModelIndex*) – an index to a parameter_value in parent_model
- **parent** (*QWidget, optional*) – a parent widget

_set_data(*value*)

See base class.

spinetoolbox.widgets.code_text_edit

Provides simple text editor for programming purposes.

Module Contents

Classes

CodeTextEdit	A plain text edit with syntax highlighting and line numbers.
LineNumberArea	

class spinetoolbox.widgets.code_text_edit.**CodeTextEdit**(**arg*, ***kwargs*)

Bases: PySide6.QtWidgets.QPlainTextEdit

A plain text edit with syntax highlighting and line numbers.

insertFromMimeData(*source*)

file_selected(*status*)

set_lexer_name(*lexer_name*)

setPlainText(*text*)

setDocument(*doc*)

line_number_area_width()

_update_line_number_area_width(*_new_block_count=0*)

_update_line_number_area(*rect*, *dy*)

_update_line_number_area_cursor_position()

set_enabled_with_greyed(*enabled*)

resizeEvent(*event*)

line_number_area_paint_event(*ev*)

class spinetoolbox.widgets.code_text_edit.**LineNumberArea**(*editor*)

Bases: PySide6.QtWidgets.QWidget

sizeHint()

paintEvent(*ev*)

spinetoolbox.widgets.commit_dialog

Classes for custom QDialogs to add edit and remove database items.

Module Contents

Classes

CommitDialog	A dialog to query user's preferences for new commit.

class spinetoolbox.widgets.commit_dialog.**CommitDialog**(*parent*, **db_names*)

Bases: PySide6.QtWidgets.QDialog

A dialog to query user's preferences for new commit.

Parameters

- **parent** (*QWidget*) – the parent widget
- **db_names** (*Iterable of str*) – database names

receive_text_changed()

Called when text changes in the commit msg text edit. Enable/disable commit button accordingly.

spinetoolbox.widgets.custom_combobox

Contains custom combo box classes.

Module Contents

Classes

CustomQComboBox	A custom QComboBox for showing kernels in Settings->Tools.
ElidedCombobox	Combobox with elided text.
OpenProjectDialogComboBox	

class spinetoolbox.widgets.custom_combobox.**CustomQComboBox**

Bases: PySide6.QtWidgets.QComboBox

A custom QComboBox for showing kernels in Settings->Tools.

mouseMoveEvent(*e*)

Catch mouseMoveEvent and accept it because the comboBox popup (QListView) has mouse tracking on as default. This makes sure the comboBox popup appears in correct position and clicking on the combobox repeatedly does not move the Settings window.

class spinetoolbox.widgets.custom_combobox.**ElidedCombobox**

Bases: PySide6.QtWidgets.QComboBox

Combobox with elided text.

paintEvent(*event*)

class spinetoolbox.widgets.custom_combobox.**OpenProjectDialogComboBox**

Bases: PySide6.QtWidgets.QComboBox

keyPressEvent(*e*)

Interrupts Enter and Return key presses when QComboBox is in focus. This is needed to prevent showing the 'Not a valid Spine Toolbox project' Notifier every time Enter is pressed.

Parameters
e (*QKeyEvent*) – Received key press event.

spinetoolbox.widgets.custom_delegates

Custom item delegates.

Module Contents

Classes

ComboBoxDelegate	

CheckBoxDelegate	A delegate that places a fully functioning QCheckBox.
RankDelegate	A delegate that places a QCheckBox but draws a number instead of the check.

class spinetoolbox.widgets.custom_delegates.**ComboBoxDelegate**(*items*)

Bases: PySide6.QtWidgets.QStyledItemDelegate

createEditor(*parent*, *option*, *index*)

paint(*painter*, *option*, *index*)

setEditorData(*editor*, *index*)

setModelData(*editor*, *model*, *index*)

updateEditorGeometry(*editor*, *option*, *index*)

_finalize_editing(*editor*)

class spinetoolbox.widgets.custom_delegates.**CheckBoxDelegate**(*parent*, *centered=True*)

Bases: PySide6.QtWidgets.QStyledItemDelegate

A delegate that places a fully functioning QCheckBox.

Parameters

- **parent** (*QWiget*) –
- **centered** (*bool*) – whether or not the checkbox should be center-aligned in the widget

data_committed

createEditor(*parent*, *option*, *index*)

Important, otherwise an editor is created if the user clicks in this cell. ** Need to hook up a signal to the model.

paint(*painter*, *option*, *index*)

Paint a checkbox without the label.

static _do_paint(*painter*, *checkbox_style_option*, *index*)

editorEvent(*event*, *model*, *option*, *index*)

Change the data in the model and the state of the checkbox when user presses left mouse button and this cell is editable. Otherwise do nothing.

setModelData(*editor*, *model*, *index*)

Do nothing. Model data is updated by handling the *data_committed* signal.

get_checkbox_rect(*option*)

class spinetoolbox.widgets.custom_delegates.**RankDelegate**(*parent*, *centered=True*)

Bases: *CheckBoxDelegate*

A delegate that places a QCheckBox but draws a number instead of the check.

Parameters

- **parent** (*QWiget*) –
- **centered** (*bool*) – whether or not the checkbox should be center-aligned in the widget

static _do_paint(*painter*, *checkbox_style_option*, *index*)

spinetoolbox.widgets.custom_menus

Classes for custom context menus and pop-up menus.

Module Contents

Classes

CustomContextMenu	Context menu master class for several context menus.
OpenProjectDialogComboBoxContextMenu	Context menu master class for several context menus.
CustomPopupMenu	Popup menu master class for several popup menus.
ItemSpecificationMenu	Context menu class for item specifications.
RecentProjectsPopupMenu	Recent projects menu embedded to 'File-Open recent' QAction.
KernelsPopupMenu	Menu embedded into 'Consoles->Start Jupyter Console' QMenu.
FilterMenuBase	Filter menu.

class spinetoolbox.widgets.custom_menus.**CustomContextMenu**(*parent*, *position*)

Bases: PySide6.QtWidgets.QMenu

Context menu master class for several context menus.

Parameters

- **parent** (*QWidget*) – Parent for menu widget (ToolboxUI)
- **position** (*QPoint*) – Position on screen

add_action(*text*, *icon=QIcon()*, *enabled=True*)

Adds an action to the context menu.

Parameters

- **text** (*str*) – Text description of the action
- **icon** (*QIcon*) – Icon for menu item
- **enabled** (*bool*) – Is action enabled?

set_action(*option*)

Sets the action which was clicked.

Parameters
option (*str*) – string with the text description of the action

get_action()

Returns the clicked action, a string with a description.

class spinetoolbox.widgets.custom_menus.**OpenProjectDialogComboBoxContextMenu**(*parent*, *position*)

Bases: *CustomContextMenu*

Context menu master class for several context menus.

Parameters

- **parent** (*QWidget*) – Parent for menu widget
- **position** (*QPoint*) – Position on screen

class spinetoolbox.widgets.custom_menus.**CustomPopupMenu**(*parent*)

Bases: PySide6.QtWidgets.QMenu

Popup menu master class for several popup menus.

Parameters
parent (*QWidget*) – Parent widget of this pop-up menu

add_action(*text*, *slot*, *enabled=True*, *tooltip=None*, *icon=None*)

Adds an action to the popup menu.

Parameters

- **text** (*str*) – Text description of the action
- **slot** (*method*) – Method to connect to action's triggered signal
- **enabled** (*bool*) – Is action enabled?
- **tooltip** (*str*) – Tool tip for the action
- **icon** (*QIcon*) – Action icon

class spinetoolbox.widgets.custom_menus.**ItemSpecificationMenu**(*toolbox*, *index*, *item=None*)

Bases: *CustomPopupMenu*

Context menu class for item specifications.

Parameters

- **toolbox** (ToolboxUI) – Toolbox that requests this menu, used as parent.
- **index** (*QModelIndex*) – the index

- **item** (ProjectItem, *optional*) – passed to show_specification_form

class spinetoolbox.widgets.custom_menus.**RecentProjectsPopupMenu**(*parent*)

Bases: *CustomPopupMenu*

Recent projects menu embedded to 'File-Open recent' QAction.

Parameters
parent (*QWidget*) – Parent widget of this menu (ToolboxUI)

has_recents()

Returns True if recent projects available, False otherwise.

add_recent_projects()

Reads the previous project names and paths from QSettings. Adds them to the QMenu as QActions.

call_clear_recents(*checked*)

Slot for Clear recents menu item.

Parameters
checked (*bool*) – Argument sent by triggered signal

call_open_project(*checked*, *p*)

Slot for catching the user selected action from the recent projects menu.

Parameters

- **checked** (*bool*) – Argument sent by triggered signal
- **p** (*str*) – Full path to a project file

class spinetoolbox.widgets.custom_menus.**KernelsPopupMenu**(*parent*)

Bases: *CustomPopupMenu*

Menu embedded into 'Consoles->Start Jupyter Console' QMenu.

Parameters
parent (*QWidget*) – Parent widget of this menu (ToolboxUI)

add_kernel(*kernel_name*, *resource_dir*, *cond*, *ico*, *deats*)

Adds a kernel entry as an action to this menu.

call_open_console(*checked*, *kernel_name*, *icon*, *conda*)

Slot for catching the user selected action from the kernel's menu.

Parameters

- **checked** (*bool*) – Argument sent by triggered signal
- **kernel_name** (*str*) – Kernel name to launch
- **icon** (*QIcon*) – Icon representing the kernel language
- **conda** (*bool*) – Is this a Conda kernel spec?

class spinetoolbox.widgets.custom_menus.**FilterMenuBase**(*parent*)

Bases: PySide6.QtWidgets.QMenu

Filter menu.

Parameters
parent (*QWidget*) – a parent widget

_set_up(*make_filter_model*, **args*, ***kwargs*)

connect_signals()

add_items_to_filter_list(*items*)

remove_items_from_filter_list(*items*)

_clear_filter()

_check_filter()

_change_filter()

abstract emit_filter_changed(*valid_values*)

spinetoolbox.widgets.custom_qgraphicsscene

Custom QGraphicsScene used in the Design View.

Module Contents

Classes

CustomGraphicsScene	A custom QGraphicsScene. It provides signals to notify about items,
DesignGraphicsScene	A scene for the Design view.

class spinetoolbox.widgets.custom_qgraphicsscene.**CustomGraphicsScene**

Bases: PySide6.QtWidgets.QGraphicsScene

A custom QGraphicsScene. It provides signals to notify about items, and a method to center all items in the scene.

At the moment it's used by DesignGraphicsScene and the GraphViewMixin

item_move_finished

Emitted when an item has finished moving.

center_items()

Centers toplevel items in the scene.

class spinetoolbox.widgets.custom_qgraphicsscene.**DesignGraphicsScene**(*parent*, *toolbox*)

Bases: *CustomGraphicsScene*

A scene for the Design view.

Mainly, it handles drag and drop events of ProjectItemDragMixin sources.

Parameters

- **parent** (*QObject*) – scene's parent object
- **toolbox** (ToolboxUI) – reference to the main window

link_about_to_be_drawn

link_drawing_finished

clear_icons_and_links()

mouseMoveEvent(*event*)

Moves link drawer.

mousePressEvent(*event*)

Puts link drawer to sleep and logs message if it looks like the user doesn't know what they're doing.

mouseReleaseEvent(*event*)

Makes link if drawer is released over a valid connector button or cancel link drawing on right button.

_finish_link()

emit_connection_failed()

keyPressEvent(*event*)

Puts link drawer to sleep if user presses ESC.

connect_signals()

Connect scene signals.

project_item_icons()

handle_selection_changed()

Activates items or links based on currently selected items (or links).

set_bg_color(*color*)

Change background color when this is changed in Settings.

Parameters
color (*QColor*) – Background color

set_bg_choice(*bg_choice*)

Set background choice when this is changed in Settings.

Parameters
bg (*str*) – "grid", "tree", or "solid"

dragLeaveEvent(*event*)

Accepts event.

dragEnterEvent(*event*)

Accept event. Then call the super class method only if drag source is not a ProjectItemDragMixin.

dragMoveEvent(*event*)

Accept event. Then call the super class method only if drag source is not a ProjectItemDragMixin.

dropEvent(*event*)

Only accept drops when the source is an instance of ProjectItemDragMixin. Capture text from event's mimedata and show the appropriate 'Add Item form.'

event(*event*)

Accepts GraphicsSceneHelp events without doing anything, to not interfere with our usage of QToolTip.showText in graphics_items.ExclamationIcon.

drawBackground(*painter*, *rect*)

Reimplemented method to make a custom background.

Parameters

- **painter** (*QPainter*) – Painter that is used to paint background
- **rect** (*QRectF*) – The exposed (viewport) rectangle in scene coordinates

_draw_solid_bg(*painter*, *rect*)

Draws solid bg.

_draw_grid_bg(*painter*, *rect*)

Draws grid bg.

_draw_tree_bg(*painter*, *rect*)

Draws 'tree of life' bg.

select_link_drawer(*drawer_type*)

Selects current link drawer.

Parameters

drawer_type (LinkType) – selected link drawer's type

spinetoolbox.widgets.custom_qgraphicsviews

Classes for custom QGraphicsViews for the Design and Graph views.

Module Contents

Classes

CustomQGraphicsView	Super class for Design and Entity QGraphicsViews.
DesignQGraphicsView	QGraphicsView for the Design View.

Functions

_fake_left_button_event(mouse_event)	Makes a left-click mouse event that is otherwise close of given event.

class spinetoolbox.widgets.custom_qgraphicsviews.**CustomQGraphicsView**(*parent*)

Bases: PySide6.QtWidgets.QGraphicsView

Super class for Design and Entity QGraphicsViews.

Parameters

parent (*QWidget*) – parent widget

abstract property _qsettings

property zoom_factor

DRAG_MIN_DURATION = 150

reset_zoom()

Resets zoom to the default factor.

keyPressEvent(*event*)

Enables zooming with plus and minus keys (comma resets zoom).

Parameters
event (*QKeyEvent*) – key press event

mousePressEvent(*event*)

Sets rubber band selection mode if Control or right mouse button is pressed. Enables resetting the zoom factor from the middle mouse button.

mouseMoveEvent(*event*)

mouseReleaseEvent(*event*)

Reestablish scroll hand drag mode.

_scroll_scene_by(*dx*, *dy*)

_use_smooth_zoom()

_drag_duration_passed(*mouse_event*)

Test is drag duration has passed.

Parameters
mouse_event (*QMouseEvent*) – current mouse event

wheelEvent(*event*)

Zooms in/out.

Parameters
event (*QWheelEvent*) – Mouse wheel event

resizeEvent(*event*)

Updates zoom if needed when the view is resized.

Parameters
event (*QResizeEvent*) – a resize event

setScene(*scene*)

Sets a new scene to this view.

Parameters
scene (DesignGraphicsScene) – a new scene

_handle_item_move_finished(*item*)

_update_zoom_limits()

Updates the minimum zoom limit and the zoom level with which the view fits all the items in the scene.

abstract _compute_max_zoom()

_handle_zoom_time_line_advanced(*pos*)

Performs zoom whenever the smooth zoom time line advances.

_handle_transformation_time_line_finished()

Cleans up after the smooth transformation time line finishes.

_handle_resize_time_line_finished()

Cleans up after resizing time line finishes.

zoom_in()

Perform a zoom in with a fixed scaling.

zoom_out()

Perform a zoom out with a fixed scaling.

gentle_zoom(*factor*, *zoom_focus=None*)

Perform a zoom by a given factor.

Parameters

- **factor** (*float*) – a scaling factor relative to the current scene scaling
- **zoom_focus** (*QPoint*) – focus of the zoom, e.g. mouse pointer position

_zoom(*factor*)

_get_viewport_scene_rect()

Returns the viewport rect mapped to the scene.

Returns
QRectF

_ensure_item_visible(*item*)

Resets zoom if item is not visible.

_set_preferred_scene_rect()

Sets the scene rect to the result of uniting the scene viewport rect and the items bounding rect.

disable_context_menu()

Disables the context menu.

enable_context_menu()

Enables the context menu.

class spinetoolbox.widgets.custom_qgraphicsviews.**DesignQGraphicsView**(*parent*)

Bases: *CustomQGraphicsView*

QGraphicsView for the Design View.

Parameters
parent (*QWidget*) – parent widget

property _qsettings

set_ui(*toolbox*)

Set a new scene into the Design View when app is started.

reset_zoom()

Resets zoom to the default factor.

_compute_max_zoom()

add_icon(*item_name*)

Adds project item's icon to the scene.

Parameters
item_name (*str*) – project item's name

remove_icon(*item_name*)

Removes project item's icon from scene.

Parameters
item_name (*str*) – name of the icon to remove

add_link(*src_connector*, *dst_connector*)

Pushes an AddLinkCommand to the toolbox undo stack.

Parameters

- **src_connector** (ConnectorButton) – source connector button
- **dst_connector** (ConnectorButton) – destination connector button

do_add_link(*connection*)

Adds given connection to the Design view.

Parameters
connection (*Connection*) – the connection to add

do_update_link(*updated_connection*)

Replaces a link on the Design view.

Parameters
updated_connection (*Connection*) – connection that was updated

remove_links(*links*)

Pushes a RemoveConnectionsCommand to the Toolbox undo stack.

Parameters
links (*list of Link*) – links to remove

do_remove_link(*connection*)

Removes a link from the scene.

Parameters
connection (*ConnectionBase*) – link's connection

remove_selected_links()

take_link(*link*)

Remove link, then start drawing another one from the same source connector.

add_jump(*src_connector*, *dst_connector*)

Pushes an AddJumpCommand to the Toolbox undo stack.

Parameters

- **src_connector** (ConnectorButton) – source connector button
- **dst_connector** (ConnectorButton) – destination connector button

do_add_jump(*jump*)

Adds given jump to the Design view.

Parameters
jump (*Jump*) – jump to add

do_update_jump(*updated_jump*)

Replaces a jump link on the Design view.

Parameters
updated_jump (*Jump*) – jump that was updated

do_remove_jump(*jump*)

Removes a jump from the scene.

Parameters
jump (*Jump*) – link's jump

contextMenuEvent(*event*)

Shows context menu for the blank view

Parameters
event (*QContextMenuEvent*) – Event

spinetoolbox.widgets.custom_qgraphicsviews.**_fake_left_button_event**(*mouse_event*)

Makes a left-click mouse event that is otherwise close of given event.

Parameters
mouse_event (*QMouseEvent*) – mouse event

Returns
left-click mouse event

Return type
QMouseEvent

spinetoolbox.widgets.custom_qlineedits

Contains a custom line edit.

Module Contents

Classes

PropertyQLineEdit	A custom QLineEdit for Project Item Properties.

class spinetoolbox.widgets.custom_qlineedits.**PropertyQLineEdit**

Bases: *spinetoolbox.widgets.custom_qwidgets.UndoRedoMixin*, PySide6.QtWidgets.QLineEdit

A custom QLineEdit for Project Item Properties.

setText(*text*)

Overridden to prevent the cursor going to the end whenever the user is still editing. This happens because we set the text programmatically in undo/redo implementations.

spinetoolbox.widgets.custom_qtableview

Custom QTableView classes that support copy-paste and the like.

Module Contents

Classes

CopyPasteTableView	Custom QTableView class with copy and paste methods.
AutoFilterCopyPasteTableView	Custom QTableView class with autofilter functionality.
IndexedParameterValueTableViewBase	Custom QTableView base class with copy and paste methods for indexed parameter values.
TimeSeriesFixedResolutionTableView	A QTableView for fixed resolution time series table.
IndexedValueTableView	A QTableView class with for variable resolution time series and time patterns.
ArrayTableView	Custom QTableView with copy and paste methods for single column tables.
MapTableView	Custom QTableView with copy and paste methods for map tables.

Functions

_range(selection)	Returns the top left and bottom right corners of selection.
_could_be_time_stamp(s)	Evaluates if given string could be a time stamp.
system_lc_numeric()	

Attributes

_
_NOT_TIME_STAMP

spinetoolbox.widgets.custom_qtableview._

class spinetoolbox.widgets.custom_qtableview.**CopyPasteTableView**(*parent=None*)

Bases: PySide6.QtWidgets.QTableView

Custom QTableView class with copy and paste methods.

property copy_action

property paste_action

init_copy_and_paste_actions()

Initializes copy and paste actions and connects relevant signals.

set_external_copy_and_paste_actions(*copy_action*, *paste_action*)

Sets the view to use external copy and paste actions.

Note that this doesn't connect the actions' trigger signals; the owner of the actions is responsible for handling them.

Parameters

- **copy_action** (*QAction*) – copy action
- **paste_action** (*QAction*) – paste action

delete_content(_=*False*)

Deletes content from editable indexes in current selection.

can_copy()

copy(_=*False*)

Copies current selection to clipboard in excel format.

can_paste()

paste(_=*False*)

Paste data from clipboard.

static _read_pasted_text(*text*)

Parses a tab separated CSV text table.

Parameters
text (*str*) – a CSV formatted table

Returns
a list of rows

Return type
list

paste_on_selection()

Pastes clipboard data on selection, but not beyond. If data is smaller than selection, repeat data to fit selection.

paste_normal()

Pastes clipboard data, overwriting cells if needed.

set_column_converter_for_pasting(*header*, *converter*)

_converters()

class spinetoolbox.widgets.custom_qtableview.**AutoFilterCopyPasteTableView**(*parent*)

Bases: *CopyPasteTableView*

Custom QTableView class with autofilter functionality.

Parameters
parent (*QObject*) –

setModel(*model*)

Disconnects the sectionPressed signal which seems to be connected by the super method. Otherwise pressing the header just selects the column.

Parameters
model (*QAbstractItemModel*) –

_trigger_filter_menu(_)

Shows current column's auto filter menu.

show_auto_filter_menu(*logical_index*)

Called when user clicks on a horizontal section header. Shows/hides the auto filter widget.

Parameters
logical_index (*int*) – header section index

class spinetoolbox.widgets.custom_qtableview.**IndexedParameterValueTableViewBase**(*parent=None*)

Bases: *CopyPasteTableView*

Custom QTableView base class with copy and paste methods for indexed parameter values.

copy(_=*False*)

Copies current selection to clipboard in CSV format.

Returns
True if data was copied on the clipboard, False otherwise

Return type
bool

abstract static _read_pasted_text(*text*)

Reads CSV formatted table.

abstract paste(_=*False*)

Pastes data from clipboard to selection.

class spinetoolbox.widgets.custom_qtableview.**TimeSeriesFixedResolutionTableView**(*parent=None*)

Bases: *IndexedParameterValueTableViewBase*

A QTableView for fixed resolution time series table.

paste(_=*True*)

Pastes data from clipboard.

static _read_pasted_text(*text*)

Parses the given CSV table. Parsing is locale aware.

Parameters
text (*str*) – a CSV table containing numbers

Returns
A list of floats

Return type
list of float

_paste_to_values_column(*values*, *first_row*, *paste_length*)

Pastes data to the Values column.

Parameters

- **values** (*list*) – a list of float values to paste
- **first_row** (*int*) – index of the first row where to paste
- **paste_length** (*int*) – length of the paste selection (can be different from len(values))

Returns
A tuple (list(pasted indexes), list(pasted values))

Return type
tuple

class spinetoolbox.widgets.custom_qtableview.**IndexedValueTableView**(*parent=None*)

Bases: *IndexedParameterValueTableViewBase*

A QTableView class with for variable resolution time series and time patterns.

paste(*_=False*)

Pastes data from clipboard.

_paste_two_columns(*data_indexes*, *data_values*, *first_row*, *paste_length*)

Pastes data indexes and values.

Parameters

- **data_indexes** (*list*) – a list of data indexes (time stamps/durations)
- **data_values** (*list*) – a list of data values
- **first_row** (*int*) – first row index
- **paste_length** (*int*) – selection length for pasting

Returns
a tuple (modified model indexes, modified model values)

Return type
tuple

_paste_single_column(*values*, *first_row*, *first_column*, *paste_length*)

Pastes a single column of data.

Parameters

- **values** (*list*) – a list of data to paste (data indexes or values)
- **first_row** (*int*) – first row index
- **paste_length** (*int*) – selection length for pasting

Returns
a tuple (modified model indexes, modified model values)

Return type
tuple

static _read_pasted_text(*text*)

Parses a given CSV table.

Parameters
text (*str*) – a CSV table

Returns
a tuple (data indexes, data values)

Return type
tuple

class spinetoolbox.widgets.custom_qtableview.**ArrayTableView**(*parent=None*)

Bases: *IndexedParameterValueTableViewBase*

Custom QTableView with copy and paste methods for single column tables.

copy(_=*False*)

Copies current selection to clipboard in CSV format.

Returns
True if data was copied on the clipboard, False otherwise

Return type
bool

paste(_=*False*)

Pastes data from clipboard.

static _read_pasted_text(*text*)

Reads the first column of given CSV table.

Parameters
text (*str*) – a CSV table

Returns
data column

Return type
list of str

class spinetoolbox.widgets.custom_qtableview.**MapTableView**(*parent=None*)

Bases: *CopyPasteTableView*

Custom QTableView with copy and paste methods for map tables.

copy(_=*False*)

Copies current selection to clipboard in Excel compatible CSV format.

Returns
True if data was copied on the clipboard, False otherwise

Return type
bool

delete_content(_=*False*)

Deletes content in current selection.

paste(_=*False*)

Pastes data from clipboard.

Returns
True if data was pasted successfully, False otherwise

Return type
bool

static _read_pasted_text(*text*)

Parses a given CSV table.

Parameters
text (*str*) – a CSV table

Returns
a list of table rows

Return type
list of list

spinetoolbox.widgets.custom_qtableview.**_range**(*selection*)

Returns the top left and bottom right corners of selection.

Parameters
selection (*QItemSelection*) – a list of selected QItemSelection objects

Returns
a tuple (top row, bottom row, left column, right column)

Return type
tuple of ints

spinetoolbox.widgets.custom_qtableview.**_NOT_TIME_STAMP**

spinetoolbox.widgets.custom_qtableview.**_could_be_time_stamp**(*s*)

Evaluates if given string could be a time stamp.

This is to deal with special cases that are not intended as time stamps but could end up as one by the very greedy `DateTime` constructor.

Parameters
s (*str*) – string to evaluate

Returns
True if s could be a time stamp, False otherwise

Return type
bool

spinetoolbox.widgets.custom_qtableview.**system_lc_numeric**()

spinetoolbox.widgets.custom_qtextbrowser

Class for a custom QTextBrowser for showing the logs and tool output.

Module Contents

Classes

CustomQTextBrowser	Custom QTextBrowser class.
MonoSpaceFontTextBrowser	Custom QTextBrowser class.

class spinetoolbox.widgets.custom_qtextbrowser.**CustomQTextBrowser**(*parent*)

Bases: `PySide6.QtWidgets.QTextBrowser`

Custom QTextBrowser class.

Parameters
parent (*QWidget*) – Parent widget

_ALL_RUNS = 'All executions'

set_toolbox(*toolbox*)

append(*text*)

Appends new text block to the end of the *original* document.

If the document contains more text blocks after the addition than a set limit, blocks are deleted at the start of the contents.

Parameters
text (*str*) – text to add

contextMenuEvent(*event*)

Reimplemented method to add a clear action into the default context menu.

Parameters
event (*QContextMenuEvent*) – Received event

clear()

_populate_executions_menu()

reset_executions_button_text()

_select_execution(*action*)

static _make_log_entry_title(*title*)

make_log_entry_point(*timestamp*)

Creates cursors (log entry points) for given items in event log.

Parameters
timestamp (*str*) – time stamp

add_log_message(*item_name*, *filter_id*, *message*)

Adds a message to an item's execution log.

Parameters

- **item_name** (*str*) – item name
- **filter_id** (*str*) – filter identifier
- **message** (*str*) – formatted message

execution_timestamps()

select_all_executions()

select_execution(*timestamp*)

_set_execution_visible(*timestamp*, *visible*)

set_item_log_selected(*selected*)

class spinetoolbox.widgets.custom_qtextbrowser.MonoSpaceFontTextBrowser(*parent*)

Bases: *CustomQTextBrowser*

Custom QTextBrowser class.

Parameters
parent (*QWidget*) – Parent widget

spinetoolbox.widgets.custom_qtreeview

Classes for custom QTreeView.

Module Contents

Classes

CopyPasteTreeView	Custom QTreeView class with copy and paste support.
SourcesTreeView	Custom QTreeView class for 'Sources' in Tool specification editor widget.
CustomTreeView	Custom QTreeView class for Tool specification editor form to enable keyPressEvent.

class spinetoolbox.widgets.custom_qtreeview.**CopyPasteTreeView**(*parent*)

Bases: PySide6.QtWidgets.QTreeView

Custom QTreeView class with copy and paste support.

Parameters
parent (*QWidget*) – parent widget

can_copy()

Returns True if tree view has a selection to copy from.

Returns
True if there is something to copy

Return type
bool

can_paste()

Returns whether it is possible to paste into this view.

Returns
True if pasting is possible, False otherwise

Return type
bool

copy()

Copy current selection to clipboard.

The default implementation copies the data as linefeed separated list.

Returns
True if data was successfully copied, False otherwise

Return type
bool

paste()

Pastes data to the view.

class spinetoolbox.widgets.custom_qtreeview.**SourcesTreeView**(*parent*)

Bases: PySide6.QtWidgets.QTreeView

Custom QTreeView class for 'Sources' in Tool specification editor widget.

Parameters
parent (*QWidget*) – parent widget

files_dropped

del_key_pressed

dragEnterEvent(*event*)

Accept file and folder drops from the filesystem.

dragMoveEvent(*event*)

Accept event.

dropEvent(*event*)

Emit files_dropped signal with a list of files for each dropped url.

keyPressEvent(*event*)

Overridden method to make the view support deleting items with a delete key.

class spinetoolbox.widgets.custom_qtreeview.**CustomTreeView**(*parent*)

Bases: PySide6.QtWidgets.QTreeView

Custom QTreeView class for Tool specification editor form to enable keyPressEvent.

Parameters
parent (*QWidget*) – The parent of this view

del_key_pressed

keyPressEvent(*event*)

Overridden method to make the view support deleting items with a delete key.

spinetoolbox.widgets.custom_qwidgets

Custom QWidgets for Filtering and Zooming.

Module Contents

Classes

ElidedTextMixin	
UndoRedoMixin	
FilterWidget	Filter widget class.
CustomWidgetAction	A QWidgetAction with custom hovering.
ToolBarWidgetAction	An action with a tool bar.
ToolBarWidgetBase	A toolbar on the right, with enough space to print a text beneath.
ToolBarWidget	A toolbar on the right, with enough space to print a text beneath.
MenuItemToolBarWidget	A menu item with a toolbar on the right.
_MenuToolBar	A custom tool bar for `MenuItemToolBarWidget`.
TitleWidgetAction	A titled separator.
WrapLabel	A QLabel that always wraps text.
HyperTextLabel	A QLabel that supports hyperlinks.
QWizardProcessPage	A QWizards page with a log. Useful for pages that need to capture the output of a process.
LabelWithCopyButton	A read only QLabel with a QToolButton that copies the text to clipboard.
ElidedLabel	A QLabel with elided text.
HorizontalSpinBox	
PropertyQSpinBox	A spinbox where undo and redo key strokes apply to the project.
SelectDatabaseItemsDialog	Dialog that lets selecting database items.
PurgeSettingsDialog	Dialog that lets selecting database items.

class spinetoolbox.widgets.custom_qwidgets.**ElidedTextMixin**(**args*, ***kwargs*)

setText(*text*)

_update_text(*text*)

_set_text_elided(*width=None*)

_elided_offset()

text()

resizeEvent(*event*)

class spinetoolbox.widgets.custom_qwidgets.**UndoRedoMixin**

keyPressEvent(*e*)

Overridden to catch and pass on the Undo and Redo commands when this line edit has the focus.

Parameters
e (*QKeyEvent*) – Event

class spinetoolbox.widgets.custom_qwidgets.**FilterWidget**(*parent*, *make_filter_model*, **args*, ***kwargs*)

Bases: PySide6.QtWidgets.QWidget

Filter widget class.

Init class.

Parameters

- **parent** (*QWidget, optional*) – parent widget
- **make_filter_model** (*Callable*) – callable that constructs the filter model
- ***args** – arguments forwarded to make_filter_model
- ****kwargs** – keyword arguments forwarded to make_filter_model

okPressed

cancelPressed

set_filter_list(*items*)

connect_signals()

save_state()

Saves the state of the FilterCheckboxListModel.

reset_state()

Sets the state of the FilterCheckboxListModel to saved state.

clear_filter()

Selects all items in FilterCheckBoxListModel.

has_filter()

Returns true if any item is filtered in FilterCheckboxListModel false otherwise.

_apply_filter()

Apply current filter and save state.

_cancel_filter()

Cancel current edit of filter and set the state to the stored state.

_filter_list()

Filter list with current text.

_text_edited(*new_text*)

Callback for edit text, starts/restarts timer. Start timer after text is edited, restart timer if text is edited before last time out.

class spinetoolbox.widgets.custom_qwidgets.**CustomWidgetAction**(*parent=None*)

Bases: PySide6.QtWidgets.QWidgetAction

A QWidgetAction with custom hovering.

Class constructor.

Parameters

parent (*QMenu*) – the widget's parent

_handle_hovered()

Hides other menus that might be shown in the parent widget and repaints it. This is to emulate the behavior of QAction.

class spinetoolbox.widgets.custom_qwidgets.**ToolBarWidgetAction**(*text*, *parent=None*, *compact=False*)

Bases: *CustomWidgetAction*

An action with a tool bar.

tool_bar

Type
QToolBar

Class constructor.

Parameters
parent (*QMenu*) – the widget's parent

eventFilter(*obj*, *ev*)

_handle_hovered()

Hides other menus that might be shown in the parent widget and repaints it. This is to emulate the behavior of QAction.

class spinetoolbox.widgets.custom_qwidgets.**ToolBarWidgetBase**(*text*, *parent=None*, *io_files=None*)

Bases: PySide6.QtWidgets.QWidget

A toolbar on the right, with enough space to print a text beneath.

tool_bar

Type
QToolBar

Class constructor.

Parameters

- **text** (*str*) –
- **parent** (*QWidget*) – the widget's parent

class spinetoolbox.widgets.custom_qwidgets.**ToolBarWidget**(*text*, *parent=None*, *io_files=None*)

Bases: *ToolBarWidgetBase*

A toolbar on the right, with enough space to print a text beneath.

tool_bar

Type
QToolBar

Class constructor.

Parameters

- **text** (*str*) –
- **parent** (*QWidget*) – the widget's parent

class spinetoolbox.widgets.custom_qwidgets.**MenuItemToolBarWidget**(*text*, *parent=None*, *compact=False*)

Bases: *ToolBarWidgetBase*

A menu item with a toolbar on the right.

Class constructor.

Parameters

- **text** (*str*) –
- **parent** (*QWidget*) – the widget's parent
- **compact** (*bool*) – if True, the widget uses the minimal space

paintEvent(*event*)

Draws the menu item, then calls the super() method to draw the tool bar.

class spinetoolbox.widgets.custom_qwidgets.**_MenuToolBar**(*parent=None*)

Bases: PySide6.QtWidgets.QToolBar

A custom tool bar for MenuItemToolBarWidget.

enabled_changed

_align_buttons()

Align all buttons to bottom so frames look good.

add_frame(*left*, *right*, *title*)

Add frame around given actions, with given title.

Parameters

- **left** (*QAction*) –
- **right** (*QAction*) –
- **title** (*str*) –

is_enabled()

addActions(*actions*)

Overriden method to customize tool buttons.

addAction(**args*, ***kwargs*)

Overriden method to customize the tool button.

sizeHint()

Make room for frames if needed.

paintEvent(*ev*)

Paint the frames.

_setup_action_button(*action*)

Customizes the QToolButton associated with given action:

1. Makes sure that the text honors the action's mnemonics.
2. Installs this as event filter on the button (see self.eventFilter()).

Must be called everytime an action is added to the tool bar.

Parameters
action (*QAction*) – Action to set up

actionEvent(*ev*)

Updates `self._enabled`: True if at least one non-separator action is enabled, False otherwise. Emits `self.enabled_changed` accordingly.

eventFilter(*obj*, *ev*)

Installed on each action's QToolButton. Ignores Up and Down key press events, so they are handled by the toolbar for custom navigation.

keyPressEvent(*ev*)

Navigates over the tool bar buttons.

hideEvent(*ev*)

class spinetoolbox.widgets.custom_qwidgets.**TitleWidgetAction**(*title*, *parent=None*)

Bases: *CustomWidgetAction*

A titled separator.

Class constructor.

Parameters
parent (*QMenu*) – the widget's parent

H_MARGIN = 5

V_MARGIN = 2

static _add_line(*widget*, *layout*)

isSeparator()

class spinetoolbox.widgets.custom_qwidgets.**WrapLabel**(*text=''*, *parent=None*)

Bases: `PySide6.QtWidgets.QLabel`

A QLabel that always wraps text.

class spinetoolbox.widgets.custom_qwidgets.**HyperTextLabel**(*text=''*, *parent=None*)

Bases: *WrapLabel*

A QLabel that supports hyperlinks.

class spinetoolbox.widgets.custom_qwidgets.**QWizardProcessPage**(*parent*)

Bases: `PySide6.QtWidgets.QWizardPage`

A QWizards page with a log. Useful for pages that need to capture the output of a process.

class _ExecutionManager

A descriptor that stores a QProcessExecutionManager. When `execution_finished` is emitted, it shows the button to copy the process log.

public_name

private_name

__set_name__(*owner*, *name*)

__get__(*obj*, *objtype=None*)

__set__(*obj*, *value*)

msg

msg_warning

msg_error

msg_success

msg_proc

msg_proc_error

_exec_mngr

_connect_signals()

_handle_copy_clicked(_=*False*)

_add_msg(*msg*)

_add_msg_warning(*msg*)

_add_msg_error(*msg*)

_add_msg_success(*msg*)

isComplete()

cleanupPage()

class spinetoolbox.widgets.custom_qwidgets.**LabelWithCopyButton**(*text=''*, *parent=None*)

Bases: PySide6.QtWidgets.QWidget

A read only QLabel with a QToolButton that copies the text to clipboard.

class spinetoolbox.widgets.custom_qwidgets.**ElidedLabel**(**args*, ***kwargs*)

Bases: *ElidedTextMixin*, PySide6.QtWidgets.QLabel

A QLabel with elided text.

class spinetoolbox.widgets.custom_qwidgets.**HorizontalSpinBox**(**args*, ***kwargs*)

Bases: PySide6.QtWidgets.QToolBar

valueChanged

value()

setMinimum(*minimum*)

setMaximum(*maximum*)

setValue(*value*, *strict=False*)

_dec_value()

_inc_value()

_focus_line_edit()

class spinetoolbox.widgets.custom_qwidgets.**PropertyQSpinBox**

Bases: *UndoRedoMixin*, PySide6.QtWidgets.QSpinBox

A spinbox where undo and redo key strokes apply to the project.

class spinetoolbox.widgets.custom_qwidgets.**SelectDatabaseItemsDialog**(*checked_states*, *ok_button_text=None*, *parent=None*)

Bases: PySide6.QtWidgets.QDialog

Dialog that lets selecting database items.

Parameters

- **checked_states** (*dict, optional*) – checked states for each item
- **ok_button_text** (*str, optional*) – alternative label for the OK button
- **parent** (*QWidget, optional*) – parent widget

_warn_checked_non_data_items = True

_ok_button_can_be_disabled = True

show()

Sets the OK button enabled before showing the dialog

get_checked_states()

Returns current item checked states.

Returns
mapping from database item name to checked flag

Return type
dict

_handle_check_box_state_changed(*_checked*)

class spinetoolbox.widgets.custom_qwidgets.**PurgeSettingsDialog**(*checked_states*, *ok_button_text=None*, *parent=None*)

Bases: *SelectDatabaseItemsDialog*

Dialog that lets selecting database items.

Parameters

- **checked_states** (*dict, optional*) – checked states for each item
- **ok_button_text** (*str, optional*) – alternative label for the OK button
- **parent** (*QWidget, optional*) – parent widget

_ok_button_can_be_disabled = False

spinetoolbox.widgets.datetime_editor

An editor widget for editing datetime database (relationship) parameter values.

Module Contents

Classes

DatetimeEditor	An editor widget for DateTime type parameter values.

Functions

_QDateTime_to_datetime(dt)	Converts a QDateTime object to Python's datetime.datetime type.
_datetime_to_QDateTime(dt)	Converts Python's datetime.datetime object to QDateTime.

spinetoolbox.widgets.datetime_editor.**_QDateTime_to_datetime**(*dt*)

Converts a QDateTime object to Python's datetime.datetime type.

spinetoolbox.widgets.datetime_editor.**_datetime_to_QDateTime**(*dt*)

Converts Python's datetime.datetime object to QDateTime.

class spinetoolbox.widgets.datetime_editor.**DatetimeEditor**(*parent=None*)

Bases: PySide6.QtWidgets.QWidget

An editor widget for DateTime type parameter values.

parent

a parent widget

Type
QWidget

_change_datetime(*new_datetime*)

Updates the internal DateTime value

set_value(*value*)

Sets the value to be edited.

value()

Returns the editor's current value.

spinetoolbox.widgets.duration_editor

An editor widget for editing duration database (relationship) parameter values.

Module Contents

Classes

DurationEditor	An editor widget for Duration type parameter values.

class spinetoolbox.widgets.duration_editor.**DurationEditor**(*parent=None*)

Bases: PySide6.QtWidgets.QWidget

An editor widget for Duration type parameter values.

parent

a parent widget

Type
QWidget

_change_duration()

Updates the value being edited.

set_value(*value*)

Sets the value for editing.

value()

Returns the current Duration.

spinetoolbox.widgets.indexed_value_table_context_menu

Context menus for parameter value editor widgets.

Module Contents

Classes

ContextMenuBase	Context menu base for parameter value editor tables.
ArrayTableContextMenu	Context menu for array editor tables.
IndexedValueTableContextMenu	Context menu for time series and time pattern editor tables.
MapTableContextMenu	Context menu for map editor tables.

Functions

_unique_row_ranges(selections)	Merged ranges in given selections to unique ranges.
_unique_column_ranges(selections)	Merged ranges in given selections to unique ranges.
_merge_intervals(intervals)	Merges given intervals if they overlap.

Attributes

_INSERT_SINGLE_COLUMN_AFTER
_INSERT_SINGLE_ROW_AFTER
_INSERT_MULTIPLE_COLUMNS_AFTER
_INSERT_MULTIPLE_ROWS_AFTER
_INSERT_SINGLE_COLUMN_BEFORE
_INSERT_SINGLE_ROW_BEFORE
_INSERT_MULTIPLE_COLUMNS_BEFORE
_INSERT_MULTIPLE_ROWS_BEFORE
_OPEN_EDITOR
_PLOT
_PLOT_IN_WINDOW
_REMOVE_COLUMNS
_REMOVE_ROWS
_TRIM_COLUMNS

spinetoolbox.widgets.indexed_value_table_context_menu._INSERT_SINGLE_COLUMN_AFTER = 'Insert column after'

spinetoolbox.widgets.indexed_value_table_context_menu._INSERT_SINGLE_ROW_AFTER = 'Insert row after'

spinetoolbox.widgets.indexed_value_table_context_menu._INSERT_MULTIPLE_COLUMNS_AFTER = 'Insert columns after...'

spinetoolbox.widgets.indexed_value_table_context_menu._INSERT_MULTIPLE_ROWS_AFTER = 'Insert rows after...'

spinetoolbox.widgets.indexed_value_table_context_menu._INSERT_SINGLE_COLUMN_BEFORE = 'Insert column before'

spinetoolbox.widgets.indexed_value_table_context_menu._INSERT_SINGLE_ROW_BEFORE = 'Insert row before'

spinetoolbox.widgets.indexed_value_table_context_menu._INSERT_MULTIPLE_COLUMNS_BEFORE = 'Insert columns before...'

spinetoolbox.widgets.indexed_value_table_context_menu._INSERT_MULTIPLE_ROWS_BEFORE = 'Insert rows before...'

spinetoolbox.widgets.indexed_value_table_context_menu._OPEN_EDITOR = 'Edit...'

spinetoolbox.widgets.indexed_value_table_context_menu._PLOT = 'Plot...'

spinetoolbox.widgets.indexed_value_table_context_menu._PLOT_IN_WINDOW = 'Plot in window'

spinetoolbox.widgets.indexed_value_table_context_menu._REMOVE_COLUMNS = 'Remove columns'

spinetoolbox.widgets.indexed_value_table_context_menu._REMOVE_ROWS = 'Remove rows'

spinetoolbox.widgets.indexed_value_table_context_menu._TRIM_COLUMNS = 'Trim columns'

class spinetoolbox.widgets.indexed_value_table_context_menu.**ContextMenuBase**(*table_view*, *position*)

Bases: PySide6.QtWidgets.QMenu

Context menu base for parameter value editor tables.

Parameters

- **table_view** (*QTableView*) – the view where the menu is invoked
- **position** (*QPoint*) – menu's position on the table view

_add_default_actions()

Adds default actions to the menu.

_first_row()

Returns the first selected row.

Returns
index to the first row

Return type
int

_insert_multiple_rows_after()

Prompts for row count, then inserts new rows below the current selection.

_insert_multiple_rows_before()

Prompts for row count, then inserts new rows above the current selection.

_insert_single_row_after()

Inserts a single row below the current selection.

_insert_single_row_before()

Inserts a single row above the current selection.

_last_row()

Returns the last selected row.

Returns
index to the last row

Return type
int

_prompt_row_count()

Prompts for number of rows to insert.

Returns
number of rows

Return type
int

_remove_rows()

Removes selected rows.

class spinetoolbox.widgets.indexed_value_table_context_menu.**ArrayTableContextMenu**(*editor*, *table_view*, *position*)

Bases: *ContextMenuBase*

Context menu for array editor tables.

Parameters

- **editor** (*ArrayEditor*) – array editor widget
- **table_view** (*QTableView*) – the view where the menu is invoked
- **position** (*QPoint*) – menu's position

_show_value_editor()

Opens the value element editor.

class spinetoolbox.widgets.indexed_value_table_context_menu.**IndexedValueTableContextMenu**(*table_view*, *position*)

Bases: *ContextMenuBase*

Context menu for time series and time pattern editor tables.

Parameters

- **table_view** (*QTableView*) – the view where the menu is invoked
- **position** (*QPoint*) – menu's position

class spinetoolbox.widgets.indexed_value_table_context_menu.**MapTableContextMenu**(*editor*, *table_view*, *position*)

Bases: *ContextMenuBase*

Context menu for map editor tables.

Parameters

- **editor** (MapEditor) – map editor widget
- **table_view** (*QTableView*) – the view where the menu is invoked
- **position** (*QPoint*) – table cell index

_first_column()

Returns the first selected column.

Returns
index to the first column

Return type
int

_insert_multiple_columns_after()

Prompts for column count, then inserts new columns right from the current selection.

_insert_multiple_columns_before()

Prompts for column count, then inserts new columns left from the current selection.

_insert_single_column_before()

Inserts a single column left from the current selection.

_insert_single_column_after()

Inserts a single column right from the current selection.

_last_column()

Returns the last selected column.

Returns
index to the last column

Return type
int

_prompt_column_count()

Prompts for number of column to insert.

Returns
number of columns

Return type
int

_remove_columns()

Removes selected columns

_show_value_editor()

Opens the value element editor.

_plot(*checked=False*)

Plots current indexes.

_plot_in_window(*action*)

Plots the selected cells in an existing window.

_trim_columns()

Removes excessive columns from the table.

spinetoolbox.widgets.indexed_value_table_context_menu.**_unique_row_ranges**(*selections*)

Merged ranges in given selections to unique ranges.

Parameters
selections (*list of QItemSelectionRange*) – selected ranges

Returns
a list of [first_row, last_row] ranges

Return type
list of list

spinetoolbox.widgets.indexed_value_table_context_menu.**_unique_column_ranges**(*selections*)

Merged ranges in given selections to unique ranges.

Parameters
selections (*list of QItemSelectionRange*) – selected ranges

Returns
a list of [first_row, last_row] ranges

Return type
list of list

spinetoolbox.widgets.indexed_value_table_context_menu.**_merge_intervals**(*intervals*)

Merges given intervals if they overlap.

Parameters
intervals (*list of list*) – a list of intervals in the form [first, last]

Returns
merged intervals in the form [first, last]

Return type
list of list

spinetoolbox.widgets.install_julia_wizard

Classes for custom QDialogs for julia setup.

Module Contents

Classes

_PageId	Enum where members are also (and must be) ints
InstallJuliaWizard	A wizard to install julia
JillNotFoundPage	
IntroPage	
SelectDirsPage	
InstallJuliaPage	A QWizards page with a log. Useful for pages that need to capture the output of a process.
SuccessPage	
FailurePage	

Attributes

jill_install	

spinetoolbox.widgets.install_julia_wizard.**jill_install**

class spinetoolbox.widgets.install_julia_wizard.**_PageId**

Bases: enum.IntEnum

Enum where members are also (and must be) ints

Initialize self. See help(type(self)) for accurate signature.

INTRO

SELECT_DIRS

INSTALL

SUCCESS

FAILURE

class spinetoolbox.widgets.install_julia_wizard.**InstallJuliaWizard**(*parent*)

Bases: PySide6.QtWidgets.QWizard

A wizard to install julia

Initialize class.

Parameters
parent (*QWidget*) – the parent widget (SettingsWidget)

julia_exe_selected

set_julia_exe()

accept()

class spinetoolbox.widgets.install_julia_wizard.**JillNotFoundPage**(*parent*)

Bases: PySide6.QtWidgets.QWizardPage

class spinetoolbox.widgets.install_julia_wizard.**IntroPage**(*parent*)

Bases: PySide6.QtWidgets.QWizardPage

nextId()

class spinetoolbox.widgets.install_julia_wizard.**SelectDirsPage**(*parent*)

Bases: PySide6.QtWidgets.QWizardPage

initializePage()

_select_install_dir()

_select_symlink_dir()

nextId()

class spinetoolbox.widgets.install_julia_wizard.**InstallJuliaPage**(*parent*)

Bases: *spinetoolbox.widgets.custom_qwidgets.QWizardProcessPage*

A QWizards page with a log. Useful for pages that need to capture the output of a process.

cleanupPage()

initializePage()

_handle_julia_install_finished(*ret*)

nextId()

class spinetoolbox.widgets.install_julia_wizard.**SuccessPage**(*parent*)

Bases: PySide6.QtWidgets.QWizardPage

initializePage()

nextId()

class spinetoolbox.widgets.install_julia_wizard.**FailurePage**(*parent*)

Bases: PySide6.QtWidgets.QWizardPage

initializePage()

nextId()

spinetoolbox.widgets.jump_properties_widget

Contains jump properties widget's business logic.

Module Contents

Classes

JumpPropertiesWidget	Widget for jump link properties.

class spinetoolbox.widgets.jump_properties_widget.**JumpPropertiesWidget**(*toolbox*, *base_color=None*)

Bases: *spinetoolbox.widgets.properties_widget.PropertiesWidgetBase*

Widget for jump link properties.

Parameters
toolbox (ToolboxUI) – The toolbox instance where this widget should be embedded

_load_condition_into_ui(*condition*)

_make_condition_from_ui()

_change_condition()

Stores jump condition to link.

_show_tool_spec_form(*_checked=False*)

_set_save_script_button_enabled()

_update_add_args_button_enabled(*_selected*, *_deselected*)

_do_update_add_args_button_enabled()

_update_remove_args_button_enabled(*_selected*, *_deselected*)

_do_update_remove_args_button_enabled()

_populate_cmd_line_args_model()

_push_update_cmd_line_args_command(*cmd_line_args*)

_remove_arg(*_=False*)

_add_args(*_=False*)

set_link(*jump*)

Hooks the widget to given jump link, so that user actions are reflected in the jump link's configuration.

Parameters
jump (LoggingJump) – link to hook into

unset_link()

Releases the widget from any links.

set_condition(*jump*, *condition*)

update_cmd_line_args(*jump*, *cmd_line_args*)

spinetoolbox.widgets.jupyter_console_widget

Class for a custom RichJupyterWidget that can run Tool instances.

Module Contents

Classes

JupyterConsoleWidget	Base class for all embedded console widgets that can run tool instances.

Attributes

traitlets_logger
asyncio_logger

spinetoolbox.widgets.jupyter_console_widget.**traitlets_logger**

spinetoolbox.widgets.jupyter_console_widget.**asyncio_logger**

class spinetoolbox.widgets.jupyter_console_widget.**JupyterConsoleWidget**(*toolbox*, *kernel_name*, *owner=None*)

Bases: qtconsole.rich_jupyter_widget.RichJupyterWidget

Base class for all embedded console widgets that can run tool instances.

Parameters

- **toolbox** (ToolboxUI) – QMainWindow instance
- **kernel_name** (*str*) – Kernel name to start
- **owner** (ProjectItem, *NoneType*) – Item that owns the console.

property owner_names

console_closed

request_start_kernel(*conda=False*)

Requests Spine Engine to launch a kernel manager for the given kernel_name.

Parameters
conda (*bool*) – Conda kernel or not

Returns
Path to connection file if kernel manager was launched successfully, None otherwise

Return type
str or None

release_exec_mngr_resources()

Closes _io.TextIOWrapper files.

_handle_kernel_started_msg(*msg*)

Handles the response message from KernelExecutionManager.

Parameters
msg (*dict*) – Message with item_name, type, etc. keys

Returns
Path to a connection file if engine started the requested kernel manager successfully, None otherwise.

Return type
str or None

_execute(*source*, *hidden*)

Catches exit or similar commands and closes the console immediately if user so chooses.

insert_text_to_console(*msg*)

Inserts given message to console.

Parameters
msg (*str*) – Text to insert

set_connection_file(*connection_file*)

Sets connection file obtained from engine to this console.

Parameters
connection_file (*str*) – Path to a connection file obtained from a running kernel manager.

connect_to_kernel()

Connects a local kernel client to a kernel manager running on Spine Engine.

request_restart_kernel_manager()

Restarts kernel manager on engine and connects a new kernel client to it.

request_shutdown_kernel_manager()

Sends a shutdown kernel manager request to engine.

name()

Returns console name for display purposes.

shutdown_kernel_client()

Shuts down local kernel client.

dragEnterEvent(*e*)

Rejects dropped project items.

_context_menu_make(*pos*)

Reimplemented to add actions to console context-menus.

copy_input()

Copies only input.

_show_interpreter_prompt(*number=None*)

Reimplemented for IPython-style prompts.

closeEvent(*e*)

Catches close event to shut down the kernel client and sends a signal to Toolbox to request Spine Engine to shut down the kernel manager.

spinetoolbox.widgets.kernel_editor

Widget for showing the progress of making a Julia or Python kernel.

Module Contents

Classes

KernelEditorBase	Base class for kernel editors.
MiniPythonKernelEditor	A Simple Python kernel maker. The Python executable path is passed in
MiniJuliaKernelEditor	A Simple Julia Kernel maker. The julia exe and project are passed in

Functions

format_event_message(msg_type, message[, show_datetime])	Formats message for the kernel editor text browser.
format_process_message(msg_type, message)	Formats process message for the kernel editor text browser.

class spinetoolbox.widgets.kernel_editor.**KernelEditorBase**(*parent*, *python_or_julia*)

Bases: PySide6.QtWidgets.QDialog

Base class for kernel editors.

Parameters

- **parent** (SettingsWidget) – Parent widget
- **python_or_julia** (*str*) – kernel type; valid values: “julia”, “python”

connect_signals()

Connects signals to slots.

_show_close_button(*failed=False*)

make_kernel()

abstract _do_make_kernel()

new_kernel_name()

Returns the new kernel name after it’s been created.

_solve_new_kernel_name()

Finds out the new kernel name after a new kernel has been created.

check_options(*prgm*, *kernel_name*, *display_name*, *python_or_julia*)

Checks that user options are valid before advancing with kernel making.

Parameters

- **prgm** (*str*) – Full path to Python or Julia program

- **kernel_name** (*str*) – Kernel name
- **display_name** (*str*) – Kernel display name
- **python_or_julia** (*str*) – Either 'python' or 'julia'

Returns
True if all user input is valid for making a new kernel, False otherwise

Return type
bool

abstract _python_kernel_name()

abstract _python_kernel_display_name()

_python_interpreter_name()

make_python_kernel(*checked=False*)

Makes a new Python kernel. Offers to install ipykernel package if it is missing from the selected Python environment. Overwrites existing kernel with the same name if this is ok by user.

static is_package_installed(*python_path*, *package_name*)

Checks if given package is installed to given Python environment.

Parameters

- **python_path** (*str*) – Full path to selected Python interpreter
- **package_name** (*str*) – Package name

Returns
True if installed, False if not

Return type
(bool)

start_package_install_process(*python_path*, *package_name*)

Starts installing the given package using pip.

Parameters

- **python_path** (*str*) – Full path to selected Python interpreter
- **package_name** (*str*) – Package name to install using pip

handle_package_install_process_finished(*retval*)

Handles installing package finished.

Parameters
retval (*int*) – Process return value. 0: success, !0: failure

start_kernelspec_install_process(*prgm*, *k_name*, *d_name*)

Installs kernel specifications for the given Python environment. Runs e.g. this command in QProcess

python -m ipykernel install –user –name python-X.Y –display-name PythonX.Y

Creates new kernel specs into %APPDATA%jupyterkernels. Existing directory will be overwritten.

Note: We cannot use –sys.prefix here because if we have selected to create a kernel for some other python that was used in launching the app, the kernel will be created into a location that is not discoverable by jupyter and hence not by Spine Toolbox. E.g. when sys.executable is C:Python36python.exe, and we have selected that as the python for Spine Toolbox (Settings->Tools->Python interpreter is empty), creating a

kernel with –sys-prefix creates kernel specs into C:Python36sharejupyterkernelspython-3.6. This is ok and the kernel spec is discoverable by jupyter and Spine Toolbox.

BUT when sys.executable is C:Python36python.exe, and we have selected another python for Spine Toolbox (Settings->Tools->Python interpreter is C:Python38python.exe), creating a kernel with –sys-prefix creates a kernel into C:Python38sharejupyterkernelspython-3.8-sys-prefix. This is not discoverable by jupyter nor Spine Toolbox. You would need to start the app using C:Python38python.exe to see and use that kernel spec.

Using –user option instead, creates kernel specs that are discoverable by any python that was used in starting Spine Toolbox.

Parameters

- **prgm** (*str*) – Full path to Python interpreter for which the kernel is created
- **k_name** (*str*) – Kernel name
- **d_name** (*str*) – Kernel display name

handle_kernelspec_install_process_finished(*retval*)

Handles case when the process for installing the kernel has finished.

Parameters

retval (*int*) – Process return value. 0: success, !0: failure

abstract _julia_kernel_name()

_julia_executable()

_julia_project()

make_julia_kernel(*checked=False*)

Makes a new Julia kernel. Offers to install IJulia package if it is missing from the selected Julia project. Overwrites existing kernel with the same name if this is ok by user.

_is_rebuild_ijulia_needed()

is_ijulia_installed(*program*, *project*)

Checks if IJulia is installed for the given project. Note: Trying command 'using IJulia' does not work since it automatically tries loading it from the LOAD_PATH if not it's not found in the active project.

Returns

0 when process failed to start, 1 when IJulia is installed, 2 when IJulia is not installed.

Return type

int

start_ijulia_install_process(*julia*, *project*)

Starts installing IJulia package to given Julia project.

Parameters

- **julia** (*str*) – Full path to selected Julia executable
- **project** (*str*) – Julia project (e.g. dir path or '@.', or '.')

handle_ijulia_install_finished(*ret*)

Runs when IJulia install process finishes.

Parameters

ret (*int*) – Process return value. 0: success, !0: failure

start_ijulia_rebuild_process(*program*, *project*)

Starts rebuilding IJulia.

handle_ijulia_rebuild_finished(*ret*)

Runs when IJulia rebuild process finishes.

Parameters
ret (*int*) – Process return value. 0: success, !0: failure

start_ijulia_installkernel_process(*program*, *project*, *kernel_name*)

Installs the kernel using IJulia.installkernel function. Given kernel_name is the new kernel DISPLAY name prefix. IJulia strips the whitespace and uncapitalizes this to make the kernel name automatically. Julia version is concatenated to both kernel and display names automatically (This cannot be changed).

handle_installkernel_process_finished(*retval*)

Checks whether the IJulia.installkernel process finished successfully.

Parameters
retval (*int*) – Process return value. 0: success, !0: failure

restore_dialog_dimensions()

Restore widget location, dimensions, and state from previous session.

add_message(*msg*)

Append regular message to kernel editor text browser.

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

add_success_message(*msg*)

Append message with green text color to kernel editor text browser.

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

add_error_message(*msg*)

Append message with red color to kernel editor text browser.

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

add_warning_message(*msg*)

Append message with yellow (golden) color to kernel editor text browser.

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

add_process_message(*msg*)

Writes message from stdout to kernel editor text browser.

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

add_process_error_message(*msg*)

Writes message from stderr to kernel editor text browser.

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

_save_ui()

class spinetoolbox.widgets.kernel_editor.**MiniPythonKernelEditor**(*parent*, *python_exe*)

Bases: *KernelEditorBase*

A Simple Python kernel maker. The Python executable path is passed in the constructor, then calling `make_kernel` starts the process.

Parameters

- **parent** (SettingsWidget) – Parent widget
- **python_or_julia** (*str*) – kernel type; valid values: "julia", "python"

abstract _julia_kernel_name()

_python_kernel_name()

_python_kernel_display_name()

_do_make_kernel()

handle_kernelspec_install_process_finished(*retval*)

Handles case when the process for installing the kernel has finished.

Parameters

retval (*int*) – Process return value. 0: success, !0: failure

set_kernel_name()

Retrieves Python version in a subprocess and makes a kernel name based on it.

class spinetoolbox.widgets.kernel_editor.**MiniJuliaKernelEditor**(*parent*, *julia_exe*, *julia_project*)

Bases: *KernelEditorBase*

A Simple Julia Kernel maker. The julia exe and project are passed in the constructor, then calling `make_kernel` starts the process.

Parameters

- **parent** (SettingsWidget) – Parent widget
- **python_or_julia** (*str*) – kernel type; valid values: "julia", "python"

_julia_kernel_name()

abstract _python_kernel_name()

abstract _python_kernel_display_name()

_do_make_kernel()

handle_installkernel_process_finished(*retval*)

Checks whether the IJulia.installkernel process finished successfully.

Parameters

retval (*int*) – Process return value. 0: success, !0: failure

spinetoolbox.widgets.kernel_editor.**format_event_message**(*msg_type*, *message*, *show_datetime=True*)

Formats message for the kernel editor text browser. This is a copy of helpers.format_event_message() but the colors have been edited for a text browser with a white background.

spinetoolbox.widgets.kernel_editor.**format_process_message**(*msg_type*, *message*)

Formats process message for the kernel editor text browser.

spinetoolbox.widgets.link_properties_widget

Link properties widget.

Module Contents

Classes

LinkPropertiesWidget	Widget for connection link properties.

class spinetoolbox.widgets.link_properties_widget.**LinkPropertiesWidget**(*toolbox*, *base_color=None*)

Bases: *spinetoolbox.widgets.properties_widget.PropertiesWidgetBase*

Widget for connection link properties.

Parameters
toolbox (ToolboxUI) – The toolbox instance where this widget should be embedded

set_link(*connection*)

Hooks the widget to given link, so that user actions are reflected in the link's filter configuration.

Parameters
connection (LoggingConnection) –

unset_link()

Releases the widget from any links.

_handle_auto_check_filters_state_changed(*checked*)

Updates filters' auto enabled setting.

Parameters
checked (*bool*) – True if the checkbox is checked, False otherwise

set_auto_check_filters_state(*checked*)

Sets the checked status of filter default online status check box

Parameters
checked (*bool*) – True if the checkbox is checked

_populate_filter_validation_menu()

Adds actions to filter validation menu.

Returns
menu actions

Return type
dict

_update_filter_validation_options(*checked*)

_handle_write_index_value_changed(*value*)

_handle_use_datapackage_state_changed(*_state*)

_handle_use_memory_db_state_changed(*_state*)

_handle_purge_before_writing_state_changed(*_state*)

_open_purge_settings_dialog(*_=False*)

Opens the purge settings dialog.

_handle_purge_settings_changed()

Pushes a command that sets new purge settings onto undo stack.

_clean_up_purge_settings_dialog()

Cleans things related to purge settings dialog.

load_connection_options()

_select_mutually_exclusive_filter(*label*)

set_filter_type_enabled(*filter_type*, *enabled*)

Enables or disables filter type in the tree.

Parameters

- **filter_type** (*str*) – filter type
- **enabled** (*bool*) – whether filter type is enabled

_set_filter_type_expanded(*filter_type*, *expanded*)

Expands or collapses filter type branch in the tree.

Parameters

- **filter_type** (*str*) – filter type
- **expanded** (*bool*) – True to expand the branch, False to collapse

spinetoolbox.widgets.map_editor

An editor widget for editing a map type parameter values.

Module Contents

Classes

MapEditor	A widget for editing maps.

class spinetoolbox.widgets.map_editor.**MapEditor**(*parent=None*)

Bases: PySide6.QtWidgets.QWidget

A widget for editing maps.

Parameters

parent (*QWidget, optional*) – parent widget

_convert_leaves(*_*)

_show_table_context_menu(*position*)

Opens table context menu.

Parameters
position (*QPoint*) – menu's position

set_value(*value*)

Sets the parameter_value to be edited.

value()

Returns the parameter_value currently being edited.

open_value_editor(*index*)

Opens value editor dialog for given map model index.

Parameters
index (*QModelIndex*) – index

_open_header_editor(*column*)

spinetoolbox.widgets.map_value_editor

An editor dialog for map indexes and values.

Module Contents

Classes

MapValueEditor	Dialog for editing parameter values in Map value editor.

class spinetoolbox.widgets.map_value_editor.**MapValueEditor**(*index*, *parent=None*)

Bases: *spinetoolbox.widgets.parameter_value_editor_base.ParameterValueEditorBase*

Dialog for editing parameter values in Map value editor.

Parameters

- **index** (*QModelIndex*) – an index to a parameter_value in parent_model
- **parent** (*QWidget, optional*) – a parent widget

_set_data(*value*)

See base class.

spinetoolbox.widgets.multi_tab_spec_editor

Contains the MultiTabSpecEditor class.

Module Contents

Classes

MultiTabSpecEditor	A main window that has a tab widget as its central widget.

class spinetoolbox.widgets.multi_tab_spec_editor.**MultiTabSpecEditor**(*toolbox*, *item_type*)

Bases: *spinetoolbox.widgets.multi_tab_window.MultiTabWindow*

A main window that has a tab widget as its central widget.

Parameters

- **qsettings** (*QSettings*) – Toolbox settings
- **settings_group** (*str*) – this window's settings group in `qsettings`

property new_tab_title

Title for new tabs.

_make_other()

Creates a new MultiTabWindow of this type.

Returns
new MultiTabWindow

Return type
MultiTabWindow

_make_new_tab(**args*, ***kwargs*)

Creates a new tab.

Parameters

- ***args** – positional arguments neede to make a new tab
- ****kwargs** – keyword arguments needed to make a new tab

_connect_tab_signals(*tab*)

Connects spec editor window (tab) signals.

Parameters
tab (SpecificationEditorWindowBase) – Specification editor window

Returns
True if ok, False otherwise

Return type
bool

_disconnect_tab_signals(*index*)

Disconnects signals of spec editor window (tab) in given index.

Parameters
index (*int*) – Tab index

Returns
True if ok, False otherwise

Return type
bool

show_plus_button_context_menu(*global_pos*)

Opens a context menu for the toolbar.

Parameters
global_pos (*QPoint*) – menu position on screen

spinetoolbox.widgets.multi_tab_window

Contains the MultiTabWindow and TabBarPlus classes.

Module Contents

Classes

MultiTabWindow	A main window that has a tab widget as its central widget.
TabBarPlus	Tab bar that has a plus button floating to the right of the tabs.

class spinetoolbox.widgets.multi_tab_window.**MultiTabWindow**(*qsettings*, *settings_group*)

Bases: PySide6.QtWidgets.QMainWindow

A main window that has a tab widget as its central widget.

Parameters

- **qsettings** (*QSettings*) – Toolbox settings
- **settings_group** (*str*) – this window's settings group in qsettings

property accepting_new_tabs

property new_tab_title

Title for new tabs.

_tab_slots

_other_editor_windows

abstract _make_other()

Creates a new MultiTabWindow of this type.

Returns
new MultiTabWindow

Return type
MultiTabWindow

others()

List of other MultiTabWindows of the same type.

Returns
other MultiTabWindows windows

Return type
list of MultiTabWindow

abstract _make_new_tab(**args*, ***kwargs*)

Creates a new tab.

Parameters

- ***args** – positional arguments neede to make a new tab
- ****kwargs** – keyword arguments needed to make a new tab

abstract show_plus_button_context_menu(*global_pos*)

Opens a context menu for the toolbar.

Parameters
global_pos (*QPoint*) – menu position on screen

connect_signals()

Connects window's signals.

name()

Generates name based on the current tab and total tab count.

Returns
a name

Return type
str

all_tabs()

Iterates over tab contents widgets.

Yields
QWidget – tab contents widget

add_new_tab(**args*, ***kwargs*)

Creates a new tab and adds it at the end of the tab bar.

Parameters

- ***args** – parameters forwarded to MutliTabWindow._make_new_tab()
- ****kwargs** – parameters forwarded to MultiTabwindow._make_new_tab()

Returns
True if successful, False otherwise

Return type
bool

insert_new_tab(*index*, **args*, ***kwargs*)

Creates a new tab and inserts it at the given index.

Parameters

- **index** (*int*) – insertion point index
- ***args** – parameters forwarded to MutliTabWindow._make_new_tab()
- ****kwargs** – parameters forwarded to MultiTabwindow._make_new_tab()

_add_connect_tab(*tab*, *text*)

Appends a new tab and connects signals.

Parameters

- **tab** (*QWidget*) – tab contents widget
- **text** (*str*) – appended tab title

_insert_connect_tab(*index*, *tab*, *text*)

Inserts a new tab and connects signals.

Parameters

- **index** (*int*) – insertion point index
- **tab** (*QWidget*) – tab contents widget
- **text** (*str*) – inserted tab title

_remove_disconnect_tab(*index*)

Disconnects and removes a tab.

Parameters
index (*int*) – tab index

_connect_tab(*index*)

Connects signals from a tab contents widget.

Parameters
index (*int*) – tab index

_connect_tab_signals(*tab*)

Connects signals from a tab contents widget.

Parameters
tab (*QWidget*) – tab contents widget

Returns
True if signals were connected successfully, False otherwise

Return type
bool

_disconnect_tab_signals(*index*)

Disconnects signals from given tab.

Parameters
index (*int*) – tab index

Returns
True if signals were disconnected successfully, False otherwise

Return type
bool

_handle_tab_window_title_changed(*tab*, *title*)

Updates tab's title.

Parameters

- **tab** (*QWidget*) – tab's content widget
- **title** (*str*) – new tab title; if emtpy, one will be generated

_take_tab(*index*)

Removes a tab and returns its contents.

Parameters
index (*int*) – tab index

Returns
widget the tab was holding and tab's title

Return type
tuple

move_tab(*index*, *other=None*)

Moves a tab to another MultiTabWindow.

Parameters

- **index** (*int*) – tab index
- **other** (*MultiTabWindow, optional*) – target window; if None, creates a new window

detach(*index*, *hot_spot*, *offset=0*)

Detaches the tab at given index into another MultiTabWindow window and starts dragging it.

Parameters

- **index** (*int*) –
- **hot_spot** (*QPoint*) –
- **offset** (*int*) –

start_drag(*hot_spot*, *offset=0*)

Starts dragging a detached tab.

Parameters

- **hot_spot** (*QPoint*) – The anchor point of the drag in widget coordinates.
- **offset** (*int*) – Horizontal offset of the tab in the bar.

_frame_height()

Calculates the total 'thickness' of window frame in vertical direction.

Returns
frame height

Return type
int

timerEvent(*event*)

Performs the drag, i.e., moves the window with the mouse cursor. As soon as the mouse hovers the tab bar of another MultiTabWindow, reattaches it.

mouseReleaseEvent(*event*)

Stops the drag. This only happens when the detached tab is not reattached to another window.

reattach(*index*, *tab*, *text*)

Reattaches a tab that has been dragged over this window's tab bar.

Parameters

- **index** (*int*) – Index in this widget's tab bar where the detached tab has been dragged.
- **tab** (*QWidget*) – The widget in the tab being dragged.

- **text** (*str*) – The title of the tab.

handle_close_request_from_tab()

Catches close event triggered by a QAction in tab's QToolBar. Calls QTabWidgets close tabs method to ensure that the last closed tab closes the editor window.

_close_tab(*index*)

Closes the tab at index.

Parameters
index (*int*) – tab index

set_current_tab(*tab*)

Sets the tab that is shown on the window.

Parameters
tab (*QWidget*) – tab's contents widget

make_context_menu(*index*)

Creates a context menu for given tab.

Parameters
index (*int*) – tab index

Returns
context menu or None if tab was not found

Return type
QMenu

restore_ui()

Restore UI state from previous session.

save_window_state()

Save window state parameters (size, position, state) via QSettings.

closeEvent(*event*)

class spinetoolbox.widgets.multi_tab_window.**TabBarPlus**(*parent*)

Bases: PySide6.QtWidgets.QTabBar

Tab bar that has a plus button floating to the right of the tabs.

Parameters
parent (MultiSpineDBEditor) –

plus_clicked

resizeEvent(*event*)

Sets the dimension of the plus button. Also, makes the tab bar as wide as the parent.

tabLayoutChange()

_move_plus_button()

Places the plus button at the right of the last tab.

mousePressEvent(*event*)

Registers the position of the press, in case we need to detach the tab.

mouseMoveEvent(*event*)

Detaches a tab either if the user moves beyond the limits of the tab bar, or if it's the only one.

_send_release_event(*pos*)

Sends a mouse release event at given position in local coordinates. Called just before detaching a tab.

Parameters
pos (*QPoint*) –

mouseReleaseEvent(*event*)

start_dragging(*index*)

Stars dragging the given index. This happens when a detached tab is reattached to this bar.

Parameters
index (*int*) –

index_under_mouse()

Returns the index under the mouse cursor, or None if the cursor isn't over the tab bar. Used to check for drop targets.

Returns
int or NoneType

contextMenuEvent(*event*)

spinetoolbox.widgets.notification

Contains a notification widget.

Module Contents

Classes

Notification	Custom pop-up notification widget with fade-in and fade-out effect.
ButtonNotification	A notification with a button.
LinkNotification	A notification that may have a link.
ChangeNotifier	**param parent**

class spinetoolbox.widgets.notification.**Notification**(*parent*, *txt*, *anim_duration=_FADE_IN_OUT_DURATION*, *life_span=None*, *word_wrap=True*, *corner=Qt.TopRightCorner*)

Bases: PySide6.QtWidgets.QFrame

Custom pop-up notification widget with fade-in and fade-out effect.

Parameters

- **parent** (*QWidget*) – Parent widget
- **txt** (*str*) – Text to display in notification
- **anim_duration** (*int*) – Duration of the animation in msecs
- **life_span** (*int*) – How long does the notification stays in place in msecs

- **word_wrap** (*bool*) –
- **corner** (*Qt.Corner*) –

_FADE_IN_OUT_DURATION = 500

opacity

show()

Shows widget and moves it to the selected corner of the parent widget.

get_opacity()

opacity getter.

set_opacity(*op*)

opacity setter.

update_opacity(*value*)

Updates graphics effect opacity.

start_self_destruction()

Starts fade-out animation and closing of the notification.

enterEvent(*e*)

Pauses timer as the mouse hovers the notification.

leaveEvent(*e*)

Starts self destruction after the mouse leaves the notification.

remaining_time()

class spinetoolbox.widgets.notification.**ButtonNotification**(**args*, *button_text=''*, *button_slot=None*, ***kwargs*)

Bases: *Notification*

A notification with a button.

Parameters

- **parent** (*QWidget*) – Parent widget
- **txt** (*str*) – Text to display in notification
- **anim_duration** (*int*) – Duration of the animation in msecs
- **life_span** (*int*) – How long does the notification stays in place in msecs
- **word_wrap** (*bool*) –
- **corner** (*Qt.Corner*) –

class spinetoolbox.widgets.notification.**LinkNotification**(**args*, *open_link=None*, ***kwargs*)

Bases: *Notification*

A notification that may have a link.

Parameters

- **parent** (*QWidget*) – Parent widget
- **txt** (*str*) – Text to display in notification
- **anim_duration** (*int*) – Duration of the animation in msecs
- **life_span** (*int*) – How long does the notification stays in place in msecs

- **word_wrap** (*bool*) –
- **corner** (*Qt.Corner*) –

class spinetoolbox.widgets.notification.**ChangeNotifier**(*parent*, *undo_stack*, *settings*, *settings_key*, *corner=Qt.BottomRightCorner*)

Bases: PySide6.QtCore.QObject

Parameters

- **parent** (*QWidget*) –
- **undo_stack** (*QUndoStack*) –
- **settings** (*QSettings*) –
- **settings_key** (*str*) –
- **corner** (*int*) –

_ANIMATION_LIFE_SPAN = 5000

_push_notification(*index*)

tear_down()

Tears down the notifier.

spinetoolbox.widgets.open_project_dialog

Contains a class for a widget that represents a 'Open Project Directory' dialog.

Module Contents

Classes

OpenProjectDialog	A dialog that lets user select a project to open either by choosing
CustomQFileSystemModel	Custom file system model.
DirValidator	

class spinetoolbox.widgets.open_project_dialog.**OpenProjectDialog**(*toolbox*)

Bases: PySide6.QtWidgets.QDialog

A dialog that lets user select a project to open either by choosing an old .proj file or by choosing a project directory.

Parameters

toolbox (ToolboxUI) – QMainWindow instance

set_keyboard_shortcuts()

Creates keyboard shortcuts for the 'Root', 'Home', etc. buttons.

connect_signals()

Connects signals to slots.

expand_and_resize(*p*)

Expands, resizes, and scrolls the tree view to the current directory when the file model has finished loading the path. Slot for the file model's directoryLoaded signal. The directoryLoaded signal is emitted only if the directory has not been cached already. Note, that this is only used when the open project dialog is opened

Parameters
p (*str*) – Directory that has been loaded

validator_state_changed()

Changes the combobox border color according to the current state of the validator.

current_index_changed(*i*)

Combobox selection changed. This slot is processed when a new item is selected from the drop-down list. This is not processed when new item txt is QValidotor.Intermediate.

Parameters
i (*int*) – Selected row in combobox

current_changed(*current*, *previous*)

Processed when the current item in file system tree view has been changed with keyboard or mouse. Updates the text in combobox.

Parameters

- **current** (*QModelIndex*) – Currently selected index
- **previous** (*QModelIndex*) – Previously selected index

set_selected_path(*index*)

Sets the text in the combobox as the selected path in the file system tree view.

Parameters
index (*QModelIndex*) – The index which was mouse clicked.

combobox_text_edited(*text*)

Updates selected path when combobox text is edited. Note: pressing enter in combobox does not trigger this.

selection()

Returns the selected path from dialog.

go_root(*checked=False*)

Slot for the 'Root' button. Scrolls the treeview to show and select the user's root directory.

Note: We need to expand and scroll the tree view here after setCurrentIndex just in case the directory has been loaded already.

go_home(*checked=False*)

Slot for the 'Home' button. Scrolls the treeview to show and select the user's home directory.

go_documents(*checked=False*)

Slot for the 'Documents' button. Scrolls the treeview to show and select the user's documents directory.

go_desktop(*checked=False*)

Slot for the 'Desktop' button. Scrolls the treeview to show and select the user's desktop directory.

open_project(*index*)

Opens project if index contains a valid Spine Toolbox project. Slot for the mouse doubleClicked signal. Prevents showing the 'Not a valid spine toolbox project' notification if user just wants to collapse a directory.

Parameters
index (*QModelIndex*) – File model index which was double clicked

done(*r*)

Checks that selected path exists and is a valid Spine Toolbox directory when ok button is clicked or when enter is pressed without the combobox being in focus.

Parameters
r (*int*) –

static update_recents(*entry*, *qsettings*)

Adds a new entry to QSettings variable that remembers the five most recent project storages.

Parameters

- **entry** (*str*) – Abs. path to a directory that most likely contains other Spine Toolbox Projects as well. First entry is also used as the initial path for File->New Project dialog.
- **qsettings** (*QSettings*) – Toolbox qsettings object

static remove_directory_from_recents(*p*, *qsettings*)

Removes directory from the recent project storages.

Parameters

- **p** (*str*) – Full path to a project directory
- **qsettings** (*QSettings*) – Toolbox qsettings object

show_context_menu(*pos*)

Shows the context menu for the QCombobox with a 'Clear history' entry.

Parameters
pos (*QPoint*) – Mouse position

closeEvent(*event=None*)

Handles dialog closing.

Parameters
event (*QCloseEvent*) – Close event

class spinetoolbox.widgets.open_project_dialog.**CustomQFileSystemModel**

Bases: PySide6.QtWidgets.QFileSystemModel

Custom file system model.

columnCount(*parent=QModelIndex()*)

Returns one.

class spinetoolbox.widgets.open_project_dialog.**DirValidator**(*parent=None*)

Bases: PySide6.QtGui.QValidator

validate(*txt*, *pos*)

Returns Invalid if input is invalid according to this validator's rules, Intermediate if it is likely that a little more editing will make the input acceptable and Acceptable if the input is valid.

Parameters

- **txt** (*str*) – Text to validate
- **pos** (*int*) – Cursor position

Returns
Invalid, Intermediate, or Acceptable

Return type
QValidator.State

spinetoolbox.widgets.options_dialog

Provides OptionsDialog.

Module Contents

Classes

OptionsDialog	A dialog with options.

class spinetoolbox.widgets.options_dialog.**OptionsDialog**(*parent*, *title*, *text*, *option_to_answer*, *notes=None*, *preferred=None*)

Bases: PySide6.QtWidgets.QDialog

A dialog with options.

Parameters

- **parent** (*QWidget*) – the parent widget
- **title** (*srt*) – title of the window
- **text** (*str*) – text to show to the user
- **option_to_answer** (*dict*) – mapping option string to corresponding answer to return
- **preferred** (*int,optional*) – preselected option if any

classmethod get_answer(*parent*, *title*, *text*, *option_to_answer*, *notes=None*, *preferred=None*)

_update_ok_button_enabled(*_id=None*)

spinetoolbox.widgets.parameter_value_editor

An editor dialog for editing database (relationship) parameter values.

Module Contents

Classes

ParameterValueEditor	Dialog for editing parameter values in Database editor.

class spinetoolbox.widgets.parameter_value_editor.**ParameterValueEditor**(*index*, *parent=None*, *plain=False*)

Bases: *spinetoolbox.widgets.parameter_value_editor_base.ParameterValueEditorBase*

Dialog for editing parameter values in Database editor.

Parameters

- **index** (*QModelIndex*) – an index to a parameter_value in parent_model
- **parent** (*QWidget, optional*) – a parent widget
- **plain** (*bool*) – if True, allow only plain value editing, otherwise allow all parameter types

_set_data(*value*)

See base class.

spinetoolbox.widgets.parameter_value_editor_base

A base for editor windows for editing parameter values.

Module Contents

Classes

ValueType	Enum to identify value types that use different editors.
ParameterValueEditorBase	Dialog for editing parameter values.

Attributes

_SELECTORS

class spinetoolbox.widgets.parameter_value_editor_base.**ValueType**

Bases: enum.Enum

Enum to identify value types that use different editors.

PLAIN_VALUE

MAP

TIME_SERIES_FIXED_RESOLUTION

TIME_SERIES_VARIABLE_RESOLUTION

TIME_PATTERN

ARRAY

DATETIME

DURATION

spinetoolbox.widgets.parameter_value_editor_base._SELECTORS

class spinetoolbox.widgets.parameter_value_editor_base.**ParameterValueEditorBase**(*index*, *editor_widgets*, *parent=None*)

Bases: PySide6.QtWidgets.QWidget

Dialog for editing parameter values.

The dialog takes an index and shows a specialized editor corresponding to the value type in a stack widget. The user can change the value type by changing the specialized editor using a combo box. When the dialog is closed the value from the currently shown specialized editor is written back to the given index.

Parameters

- **index** (*QModelIndex*) – an index to a parameter_value in parent_model
- **editor_widgets** (*dict*) – a mapping from *ValueType* to QWidget
- **parent** (*QWidget, optional*) – a parent widget

accept()

Saves the parameter_value shown in the currently selected editor widget to the database manager.

_change_parameter_type(*selector_index*)

Handles switching between value types.

Does a rude conversion between fixed and variable resolution time series. In other cases, a default 'empty' value is used.

Parameters
selector_index (*int*) – an index to the selector combo box

_select_editor(*value*)

Shows the editor widget corresponding to the given value type on the editor stack.

_use_default_editor(*message=None*)

Opens the default editor widget. Optionally, displays a warning dialog indicating the problem.

Parameters
message (*str, optional*) –

_use_editor(*value*, *value_type*)

Sets a value to edit on an editor widget.

Parameters

- **value** (*object*) – value to edit
- **value_type** (ValueType) – type of value

abstract _set_data(*value*)

Writes parameter value back to the model.

Parameters
value (*object*) – value to write

Returns
True if the operation was successful, False otherwise

Return type
bool

spinetoolbox.widgets.persistent_console_widget

Contains a widget acting as a console for Julia & Python REPL's.

Module Contents

Classes

_CustomLineEdit	
PersistentConsoleWidget	A widget to interact with a persistent process.
AnsiEscapeCodeHandler	

Functions

_ansi_color(code[, bright])	

class spinetoolbox.widgets.persistent_console_widget.**_CustomLineEdit**(*console*)

Bases: PySide6.QtWidgets.QPlainTextEdit

property min_pos

property new_line_indent

reset(*current_prompt*)

new_line()

formatted_text()

raw_text()

set_raw_text(*text*)

_handle_text_changed()

Add indent to new lines.

_handle_cursor_position_changed()

Move cursor away from indent areas.

keyPressEvent(*ev*)

class spinetoolbox.widgets.persistent_console_widget.**PersistentConsoleWidget**(*toolbox*, *key*, *language*, *owner=None*)

Bases: PySide6.QtWidgets.QPlainTextEdit

A widget to interact with a persistent process.

Parameters

- **toolbox** (ToolboxUI) –
- **key** (*tuple*) – persistent process identifier
- **language** (*str*) – for syntax highlighting and prompting, etc.
- **owner** (*ProjectItemBase, optional*) – console owner

property prompt

property owner_names

property _input_start_pos

_command_checked

_msg_available

_command_finished

_history_item_available

_completions_available

_restarted

_killed

_flush_needed

_FLUSH_INTERVAL = 200

_MAX_LINES_PER_SECOND = 2000

_MAX_LINES_PER_CYCLE

_MAX_LINES_COUNT = 2000

closeEvent(*ev*)

name()

Returns console name for display purposes.

focusInEvent(*ev*)

mouseMoveEvent(*ev*)

mousePressEvent(*ev*)

mouseReleaseEvent(*ev*)

scrollContentsBy(*dx*, *dy*)

_handle_contents_changed()

_handle_selection_changed()

_handle_cursor_position_changed()

_handle_update_request(*_rect*, *_dy*)

Move line edit to input start pos.

resizeEvent(*ev*)

_move_and_resize_line_edit()

_update_user_input()

_start_flush_timer()

_flush_text_buffer()

Inserts all text from buffer.

_make_prompt()

_make_prompt_block(*prompt=''*)

_insert_prompt(*prompt=''*)

_insert_stdin_text(*cursor*, *text*)

Inserts highlighted text.

Parameters

- **cursor** (*QTextCursor*) –
- **text** (*str*) –

_do_insert_stdin_text(*cursor*, *text*)

_insert_stdout_text(*cursor*, *text*)

Inserts ansi highlighted text.

Parameters

- **cursor** (*QTextCursor*) –
- **text** (*str*) –

_insert_text_before_prompt(*text*, *with_prompt=False*)

Inserts given text before the prompt. Used when adding input and output from external execution.

Parameters

text (*str*) –

_insert_text(*cursor*, *text*, *with_prompt*)

set_killed(*killed*)

Emits the `killed` signal.

Parameters

killed (*bool*) – if True, may the console rest in peace

_do_set_killed(*killed*)

Sets the console as killed or alive.

Parameters

killed (*bool*) – if True, may the console rest in peace

add_stdin(*data*)

Adds new prompt with data. Used when adding stdin from external execution.

Parameters

data (*str*) –

add_stdout(*data*)

Adds new line to stdout. Used when adding stdout from external execution.

Parameters
data (*str*) –

add_stderr(*data*)

Adds new line to stderr. Used when adding stderr from external execution.

Parameters
data (*str*) –

_get_current_text()

_get_prefix()

_highlight_current_input()

key_press_event(*ev*)

Handles key press event from line edit.

Returns
True if handled, False if not.

create_engine_manager()

Returns a new local or remote spine engine manager or an existing remote spine engine manager. Returns None if connecting to Spine Engine Server fails.

_issue_command(*text*)

Issues command.

Parameters
text (*str*) –

_do_check_command(*text*)

_handle_command_checked(*text*, *complete*)

Issues command.

Parameters
text (*str*) –

_do_issue_command(*text*)

_handle_msg_available(*msg_type*, *text*)

_handle_command_finished()

_move_history(*text*, *backwards*)

Moves history.

_do_move_history(*text*, *backwards*)

_display_history_item(*history_item*, *prefix*)

_autocomplete(*text*)

Autocompletes current text in the prompt (or output options if multiple matches).

Parameters
text (*str*) –

_do_autocomplete(*text*)

_display_completions(*text*, *prefix*, *completions*)

_restart_persistent(*_=False*)

Restarts underlying persistent process.

_do_restart_persistent()

_handle_restarted()

_interrupt_persistent(*_=False*)

Sends a task to executor which will interrupt the underlying persistent process.

_do_interrupt_persistent()

Interrupts the underlying persistent process.

_kill_persistent(*_=False*)

Sends a task to executor which will kill the underlying persistent process.

_do_kill_persistent()

Kills underlying persistent process.

_extend_menu(*menu*)

Appends two more actions: Restart, and Interrupt.

Parameters

menu (*QMenu*) – where to append

contextMenuEvent(*ev*)

Reimplemented to extend menu with custom actions.

class spinetoolbox.widgets.persistent_console_widget.**AnsiEscapeCodeHandler**(*fg_color*, *bg_color*)

_make_default_format()

endFormatScope()

setFormatScope(*char_format*)

parse_text(*text*)

spinetoolbox.widgets.persistent_console_widget.**_ansi_color**(*code*, *bright=False*)

spinetoolbox.widgets.plain_parameter_value_editor

An editor widget for editing plain number database (relationship) parameter values.

Module Contents

Classes

PlainParameterValueEditor	A widget to edit float or boolean type parameter values.

class spinetoolbox.widgets.plain_parameter_value_editor.**PlainParameterValueEditor**(*parent_widget=None*)

Bases: PySide6.QtWidgets.QWidget

A widget to edit float or boolean type parameter values.

Parameters
parent_widget (*QWidget*) – a parent widget

_set_number_or_string_enabled(*on*)

_set_string_enabled(*on*)

set_value(*value*)

Sets the value to be edited in this widget.

value()

Returns the value currently being edited.

spinetoolbox.widgets.plot_canvas

A Qt widget to use as a matplotlib backend.

Module Contents

Classes

LegendPosition	Generic enumeration.
PlotCanvas	A widget for plotting with matplotlib.

class spinetoolbox.widgets.plot_canvas.**LegendPosition**

Bases: enum.Enum

Generic enumeration.

Derive from this class to define new enumerations.

BOTTOM

RIGHT

class spinetoolbox.widgets.plot_canvas.**PlotCanvas**(*parent=None*, *legend_axes_position=LegendPosition.BOTTOM*)

Bases: matplotlib.backends.backend_qt5agg.FigureCanvasQTAgg

A widget for plotting with matplotlib.

Parameters

- **legend_axes_position** (LegendPosition) – legend axes position relative to plot axes
- **parent** (*QWidget, optional*) – a parent widget

property axes

figure's axes

Type
matplotlib.axes.Axes

property legend_axes

figure's legend axes

Type
matplotlib.axes.Axes

has_twinned_axes()

Checks whether the axes have been twinned.

Returns
True if axes have been twinned, False otherwise

Return type
bool

twinned_axes()

Returns twinned axes.

Returns
twinned axes

Return type
list of Axes

spinetoolbox.widgets.plot_widget

A Qt widget showing a toolbar and a matplotlib plotting canvas.

Module Contents

Classes

PlotWidget	A widget that contains a toolbar and a plotting canvas.
_PlotDataView	Custom QTableView class with copy and paste methods.
_PlotDataWidget	

Functions

prepare_plot_in_window_menu(menu)	Fills a given menu with available plot window names.

class spinetoolbox.widgets.plot_widget.**PlotWidget**(*parent=None*, *legend_axes_position=LegendPosition.BOTTOM*)

Bases: PySide6.QtWidgets.QWidget

A widget that contains a toolbar and a plotting canvas.

canvas

the plotting canvas

Type
PlotCanvas

original_xy_data

unmodified data on which the plots are based

Type
list of XYData

Parameters

- **parent** (*QWidget, optional*) – parent widget
- **legend_axes_position** (LegendPosition) – legend axes position relative to plot axes

plot_windows

A global list of plot windows.

closeEvent(*event*)

Removes the window from plot_windows and closes.

contextMenuEvent(*event*)

Shows plot context menu.

_get_plot_data()

Gathers plot data into a table.

Returns
data as table

Return type
list of list

copy_plot_data()

Copies plot data to clipboard.

show_plot_data()

Opens a separate window that shows the plot data.

add_legend(*handles*)

Adds a legend to the plot's legend axes.

Parameters
handles (*list*) – legend handles

use_as_window(*parent_window*, *document_name*)

Prepares the widget to be used as a window and adds it to plot_windows list.

Parameters

- **parent_window** (*QWidget*) – a parent window
- **document_name** (*str*) – a string to add to the window title

static _unique_window_name(*document_name*)

Returns an unique identifier for a new plot window.

class spinetoolbox.widgets.plot_widget.**_PlotDataView**(*parent=None*)

Bases: *spinetoolbox.widgets.custom_qtableview.CopyPasteTableView*

Custom QTableView class with copy and paste methods.

contextMenuEvent(*event*)

class spinetoolbox.widgets.plot_widget.**_PlotDataWidget**(*rows*, *parent=None*)

Bases: PySide6.QtWidgets.QWidget

set_size_according_to_parent()

Sets the size of the widget according to the parent widget's dimensions and the data in the table

spinetoolbox.widgets.plot_widget.**prepare_plot_in_window_menu**(*menu*)

Fills a given menu with available plot window names.

Parameters

menu (*QMenu*) – menu to modify

spinetoolbox.widgets.plugin_manager_widgets

Contains PluginManager dialogs and widgets.

Module Contents

Classes

_InstallPluginModel	
_ManagePluginsModel	
InstallPluginDialog	Initialize class
ManagePluginsDialog	Initialize class

class spinetoolbox.widgets.plugin_manager_widgets.**_InstallPluginModel**

Bases: PySide6.QtGui.QStandardItemModel

data(*index*, *role=None*)

class spinetoolbox.widgets.plugin_manager_widgets.**_ManagePluginsModel**

Bases: *_InstallPluginModel*

flags(*index*)

class spinetoolbox.widgets.plugin_manager_widgets.**InstallPluginDialog**(*parent*)

Bases: PySide6.QtWidgets.QDialog

Initialize class

item_selected

populate_list(*names*)

_handle_search_text_changed(*_text*)

_filter_model()

_handle_ok_clicked(*_=False*)

_emit_item_selected(*index*)

_update_ok_button_enabled(*_selected*, *_deselected*)

class spinetoolbox.widgets.plugin_manager_widgets.**ManagePluginsDialog**(*parent*)

Bases: PySide6.QtWidgets.QDialog

Initialize class

item_removed

item_updated

populate_list(*names*)

_create_plugin_widget(*plugin_name*, *can_update*)

_emit_item_removed(*plugin_name*)

_emit_item_updated(*plugin_name*)

spinetoolbox.widgets.project_item_drag

Classes for custom QListView.

Module Contents

Classes

ProjectItemDragMixin	Custom class with dragging support.
NiceButton	
ProjectItemButtonBase	Custom class with dragging support.
ProjectItemButton	Custom class with dragging support.
ProjectItemSpecButton	Custom class with dragging support.
ShadeMixin	
ShadeProjectItemSpecButton	Custom class with dragging support.
ShadeButton	
_ChoppedIcon	
_ChoppedIconEngine	

class spinetoolbox.widgets.project_item_drag.**ProjectItemDragMixin**(**args*, ***kwargs*)

Custom class with dragging support.

drag_about_to_start

_reset()

mousePressEvent(*event*)

mouseMoveEvent(*event*)

Start dragging action if needed

mouseReleaseEvent(*event*)

Forget drag start position

enterEvent(*event*)

class spinetoolbox.widgets.project_item_drag.**NiceButton**(**args*, ***kwargs*)

Bases: PySide6.QtWidgets.QToolButton

setText(*text*)

set_orientation(*orientation*)

class spinetoolbox.widgets.project_item_drag.**ProjectItemButtonBase**(*toolbox*, *item_type*, *icon*, *parent=None*)

Bases: *ProjectItemDragMixin*, *NiceButton*

Custom class with dragging support.

_show_tool_tip(*_=False*)

set_colored_icons(*colored*)

_handle_drag_about_to_start()

mousePressEvent(*event*)

Register drag start position

abstract _make_mime_data_text()

class spinetoolbox.widgets.project_item_drag.**ProjectItemButton**(*toolbox*, *item_type*, *icon*, *parent=None*)

Bases: *ProjectItemButtonBase*

Custom class with dragging support.

double_clicked

_make_mime_data_text()

mouseDoubleClickEvent(*event*)

class spinetoolbox.widgets.project_item_drag.**ProjectItemSpecButton**(*toolbox*, *item_type*, *icon*, *spec_name=''*, *parent=None*)

Bases: *ProjectItemButtonBase*

Custom class with dragging support.

property spec_name

_make_mime_data_text()

contextMenuEvent(*event*)

mouseDoubleClickEvent(*event*)

class spinetoolbox.widgets.project_item_drag.**ShadeMixin**

paintEvent(*ev*)

class spinetoolbox.widgets.project_item_drag.**ShadeProjectItemSpecButton**(*toolbox*, *item_type*, *icon*, *spec_name=''*, *parent=None*)

Bases: *ShadeMixin*, *ProjectItemSpecButton*

Custom class with dragging support.

clone()

class spinetoolbox.widgets.project_item_drag.**ShadeButton**(**args*, ***kwargs*)

Bases: *ShadeMixin*, *NiceButton*

class spinetoolbox.widgets.project_item_drag.**_ChoppedIcon**(*icon*, *size*)

Bases: PySide6.QtGui.QIcon

update()

class spinetoolbox.widgets.project_item_drag.**_ChoppedIconEngine**(*icon*, *size*)

Bases: PySide6.QtGui.QIconEngine

update()

pixmap(*size*, *mode*, *state*)

spinetoolbox.widgets.properties_widget

Contains PropertiesWidgetBase.

Module Contents

Classes

PropertiesWidgetBase	Properties widget base class.

class spinetoolbox.widgets.properties_widget.**PropertiesWidgetBase**(*toolbox*, *base_color=None*)

Bases: PySide6.QtWidgets.QWidget

Properties widget base class.

property fg_color

set_item(*project_item*)

Sets the active project item.

Parameters
project_item (ProjectItem) – active project item

unset_item()

Unsets the active project item.

set_color_and_icon(*base_color*, *icon=None*)

eventFilter(*obj*, *ev*)

paintEvent(*ev*)

Paints background

spinetoolbox.widgets.report_plotting_failure

Functions to report failures in plotting to the user.

Module Contents

Functions

report_plotting_failure(error, parent_widget)	Reports a PlottingError exception to the user.

spinetoolbox.widgets.report_plotting_failure.**report_plotting_failure**(*error*, *parent_widget*)

Reports a PlottingError exception to the user.

Parameters

- **error** (PlottingError) – exception to report
- **parent_widget** (*QWidget*) – parent widget

spinetoolbox.widgets.select_database_items

A widget and utilities to select database items.

Module Contents

Classes

SelectDatabaseItems	Widget that allows selecting database items.

Functions

add_check_boxes(check_boxes, checked_states, ...)	Adds check boxes to grid layout.
batch_set_check_state(boxes, checked)	Sets the checked state of multiple check boxes.

spinetoolbox.widgets.select_database_items.**add_check_boxes**(*check_boxes*, *checked_states*, *select_all_button*, *deselect_all_button*, *state_changed_slot*, *layout*)

Adds check boxes to grid layout.

Parameters

- **check_boxes** (*dict*) – mapping from label to QCheckBox
- **checked_states** (*dict*) – mapping from label to checked state boolean
- **select_all_button** (*QPushButton*) – the Select all button
- **deselect_all_button** (*QPushButton*) – the Deselect all button
- **state_changed_slot** (*Callable*) – slot to call when any checked state changes
- **layout** (*QGridLayout*) – target layout

spinetoolbox.widgets.select_database_items.**batch_set_check_state**(*boxes*, *checked*)

Sets the checked state of multiple check boxes.

Parameters

- **boxes** (*Iterable of QCheckBox*) – check boxes
- **checked** (*bool*) – checked state

class spinetoolbox.widgets.select_database_items.**SelectDatabaseItems**(*checked_states=None*, *parent=None*)

Bases: PySide6.QtWidgets.QWidget

Widget that allows selecting database items.

Parameters

- **checked_states** (*dict, optional*) – mapping from item name to check state boolean
- **parent** (*QWidget*) – parent widget

checked_state_changed

COLUMN_COUNT = 3

_DATA_ITEMS = ('entity', 'entity_group', 'parameter_value', 'entity_metadata', 'parameter_value_metadata')

_SCENARIO_ITEMS = ('alternative', 'scenario', 'scenario_alternative')

checked_states()

Collects the checked states of database items.

Returns
mapping from item name to checked state boolean

Return type
dict

any_checked()

Checks if any of the checkboxes is checked.

Returns
True if any check box is checked, False otherwise

Return type
bool

any_structural_item_checked()

_select_data_items(*_=False*)

Checks all data items.

_select_scenario_items(*_=False*)

Checks all scenario items.

spinetoolbox.widgets.set_description_dialog

A widget for editing project description.

Module Contents

Classes

SetDescriptionDialog	Dialog for setting a description for a project.

class spinetoolbox.widgets.set_description_dialog.**SetDescriptionDialog**(*toolbox*, *project*)

Bases: PySide6.QtWidgets.QDialog

Dialog for setting a description for a project.

Parameters

- **toolbox** (ToolboxUI) – QMainWindow instance
- **project** (SpineToolboxProject) –

property description

_set_ok_enabled()

accept()

spinetoolbox.widgets.settings_widget

Widget for controlling user settings.

Module Contents

Classes

SettingsWidgetBase	**param qsettings** Toolbox settings
SpineDBEditorSettingsMixin	
SpineDBEditorSettingsWidget	A widget to change user's preferred settings, but only for the Spine db editor.
SettingsWidget	A widget to change user's preferred settings.

Functions

_get_kernel_name_by_exe(p, kernel_model)	Returns the kernel name corresponding to given executable or an empty string if not found.
_selected_project_matches_kernel_project(...)	Checks if given Julia kernel's project matches the given Julia project.
_samefile(a, b)	

class spinetoolbox.widgets.settings_widget.**SettingsWidgetBase**(*qsettings*)

Bases: PySide6.QtWidgets.QWidget

Parameters
qsettings (*QSettings*) – Toolbox settings

property qsettings

connect_signals()

Connect signals.

keyPressEvent(*e*)

Close settings form when escape key is pressed.

Parameters
e (*QKeyEvent*) – Received key press event.

mousePressEvent(*e*)

Save mouse position at the start of dragging.

Parameters
e (*QMouseEvent*) – Mouse event

mouseReleaseEvent(*e*)

Save mouse position at the end of dragging.

Parameters
e (*QMouseEvent*) – Mouse event

mouseMoveEvent(*e*)

Moves the window when mouse button is pressed and mouse cursor is moved.

Parameters
e (*QMouseEvent*) – Mouse event

update_ui()

Updates UI to reflect current settings. Called when the user choses to cancel their changes. Undoes all temporary UI changes that resulted from the user playing with certain settings.

save_settings()

Gets selections and saves them to persistent memory.

update_ui_and_close()

Updates UI to reflect current settings and close.

save_and_close()

Saves settings and close.

class spinetoolbox.widgets.settings_widget.**SpineDBEditorSettingsMixin**

connect_signals()

Connect signals.

read_settings()

Read saved settings from app QSettings instance and update UI to display them.

save_settings()

Get selections and save them to persistent memory.

update_ui()

set_hide_empty_classes(*checked=False*)

set_auto_expand_entities(*checked=False*)

set_merge_dbs(*checked=False*)

set_snap_entities(*checked=False*)

set_max_entity_dimension_count(*value=None*)

set_build_iters(*value=None*)

set_spread_factor(*value=None*)

set_neg_weight_exp(*value=None*)

_set_graph_property(*name*, *value*)

class spinetoolbox.widgets.settings_widget.SpineDBEditorSettingsWidget(*multi_db_editor*)

Bases: *SpineDBEditorSettingsMixin*, *SettingsWidgetBase*

A widget to change user's preferred settings, but only for the Spine db editor.

Initialize class.

property db_mngr

show()

class spinetoolbox.widgets.settings_widget.SettingsWidget(*toolbox*)

Bases: *SpineDBEditorSettingsMixin*, *SettingsWidgetBase*

A widget to change user's preferred settings.

Parameters
toolbox (ToolboxUI) – Parent widget.

property db_mngr

connect_signals()

Connect signals.

_make_python_kernel_context_menu()

Returns a context-menu for Python kernel comboBox.

_make_julia_kernel_context_menu()

Returns a context-menu for Julia kernel comboBox.

eventFilter(*o*, *event*)

Event filter that catches mouse right button release events. This event typically closes the context-menu, but we want to prevent this and show a context-menu instead.

Parameters

- **o** (*QObject*) – Watcher
- **event** (*QEvent*) – Event

Returns
True when event is caught, False otherwise

Return type
bool

_update_python_widgets_enabled(*state*)

Enables or disables some widgets based on given boolean state.

_update_julia_widgets_enabled(*state*)

Enables or disables some widgets based on given boolean state.

_update_remote_execution_page_widget_status(*state*)

Enables or disables widgets on Remote Execution page, based on the state of remote execution enabled check box.

_show_install_julia_wizard(*_=False*)

Opens Install Julia Wizard.

_show_add_up_spine_opt_wizard(*_=False*)

Opens the add/update SpineOpt wizard.

browse_gams_button_clicked(*_=False*)

Calls static method that shows a file browser for selecting a Gams executable.

browse_julia_button_clicked(*_=False*)

Calls static method that shows a file browser for selecting a Julia path.

browse_julia_project_button_clicked(*_=False*)

Calls static method that shows a folder browser for selecting a Julia project.

browse_python_button_clicked(*_=False*)

Calls static method that shows a file browser for selecting a Python interpreter.

browse_conda_button_clicked(*_=False*)

Calls static method that shows a file browser for selecting a Conda executable.

browse_certificate_directory_clicked(*_=False*)

Calls static method that shows a file browser for selecting the security folder for Engine Server.

make_python_kernel(*_=False*)

Makes a Python kernel for Jupyter Console based on selected Python interpreter. If a kernel using this Python interpreter already exists, sets that kernel selected in the comboBox.

make_julia_kernel(*_=False*)

Makes a Julia kernel for Jupyter Console based on selected Julia executable and Julia project. If a kernel using the selected Julia executable and project already exists, sets that kernel selected in the comboBox.

show_python_kernel_context_menu_on_combobox(*pos*)

Shows the context-menu on Python kernels combobox.

show_julia_kernel_context_menu_on_combobox(*pos*)

Shows the context-menu on Julia kernels combobox.

show_python_kernel_context_menu_on_combobox_list(*pos*)

Shows the context-menu on Python kernels combobox popup list.

show_julia_kernel_context_menu_on_combobox_list(*pos*)

Shows the context-menu on Julia kernels combobox popup list.

_open_python_kernel_resource_dir(*_=False*)

Opens Python kernels resource dir.

_open_julia_kernel_resource_dir(*_=False*)

Opens Julia kernels resource dir.

open_rsc_dir(*item*)

Open path hidden in given item's tooltip in file browser.

browse_work_path(*_=False*)

Open file browser where user can select the path to wanted work directory.

show_color_dialog(*_=False*)

Lets user pick the background color from a color dialog.

update_bg_color()

Set tool button icon as the selected color and update Design View scene background color.

update_scene_bg(_=*False*)

Draw background on scene depending on radiobutton states.

update_links_geometry(*checked=False*)

update_items_path(*checked=False*)

set_toolbar_colored_icons(*checked=False*)

_update_properties_widget(*_checked=False*)

read_settings()

Read saved settings from app QSettings instance and update UI to display them.

_read_engine_settings()

Reads Engine settings and sets the corresponding UI elements.

save_settings()

Get selections and save them to persistent memory. Note: On Linux, True and False are saved as boolean values into QSettings. On Windows, booleans and integers are saved as strings. To make it consistent, we should use strings.

_save_engine_settings()

Stores Engine settings to application settings.

Returns
True if settings were stored successfully, False otherwise

Return type
bool

_get_julia_settings()

Returns current Julia execution settings in Settings->Tools widget.

set_work_directory(*new_work_dir*)

Sets new work directory.

Parameters
new_work_dir (*str*) – Possibly a new work directory

update_ui()

Updates UI to reflect current settings. Called when the user choses to cancel their changes. Undoes all temporary UI changes that resulted from the user playing with certain settings.

_edit_remote_host(*new_text*)

Prepends host line edit with the protocol for user convenience.

Parameters
new_text (*str*) – Text in the line edit after user has entered a character

start_fetching_julia_kernels()

Starts a thread for fetching Julia kernels.

stop_fetching_julia_kernels()

Terminates the kernel fetcher thread.

add_julia_kernel(*kernel_name*, *resource_dir*, *conda*, *icon*, *deats*)

Adds a kernel entry as an item to Julia kernels comboBox.

restore_saved_julia_kernel()

Sets saved or given julia kernel selected after kernels have been loaded.

start_fetching_python_kernels(*conda_path_updated=False*)

Starts a thread for fetching Python kernels.

stop_fetching_python_kernels()

Terminates the kernel fetcher thread.

add_python_kernel(*kernel_name*, *resource_dir*, *conda*, *icon*, *deats*)

Adds a kernel entry as an item to Python kernels comboBox.

restore_saved_python_kernel()

Sets saved or given python kernel selected after kernels have been loaded.

_refresh_python_kernels(*conda_path*)

Refreshes Python kernels when the conda line edit points to a valid conda executable or when the line edit is cleared.

Parameters
conda_path (*str*) – Text in line edit after it's been changed.

closeEvent(*ev*)

spinetoolbox.widgets.settings_widget.**_get_kernel_name_by_exe**(*p*, *kernel_model*)

Returns the kernel name corresponding to given executable or an empty string if not found.

Parameters

- **p** (*str*) – Absolute path to an executable
- **kernel_model** (*QStandardItemModel*) – Model containing items, which contain kernel spec details as item data

Returns
Kernel name or an empty string

Return type
str

spinetoolbox.widgets.settings_widget.**_selected_project_matches_kernel_project**(*julia_kernel_name*, *julia_project*, *kernel_model*)

Checks if given Julia kernel's project matches the given Julia project.

Parameters

- **julia_kernel_name** (*str*) – Kernel name
- **julia_project** (*str*) – Path or some other string (e.g. '@.') to denote the Julia project
- **kernel_model** (*QStandardItemModel*) – Model containing kernels

Returns
True if projects match, False otherwise

Return type
bool

spinetoolbox.widgets.settings_widget.**_samefile**(*a*, *b*)

spinetoolbox.widgets.statusbars

Functions to make and handle QStatusBars.

Module Contents

Classes

MainStatusBar	A status bar for the main toolbox window.

class spinetoolbox.widgets.statusbars.**MainStatusBar**(*toolbox*)

Bases: PySide6.QtWidgets.QStatusBar

A status bar for the main toolbox window.

Parameters
toolbox (ToolboxUI) –

_ALL_RUNS = 'All executions'

_populate_executions_menu()

reset_executions_button_text()

_select_execution(*action*)

spinetoolbox.widgets.time_pattern_editor

An editor widget for editing a time pattern type (relationship) parameter values.

Module Contents

Classes

TimePatternEditor	A widget for editing time patterns.

class spinetoolbox.widgets.time_pattern_editor.**TimePatternEditor**(*parent=None*)

Bases: PySide6.QtWidgets.QWidget

A widget for editing time patterns.

Parameters
parent (*QWidget*) – parent widget

_show_table_context_menu(*position*)

Opens the table's context menu.

Parameters
position (*QPoint*) – menu's position on the table

set_value(*value*)

Sets the parameter_value to be edited.

value()

Returns the parameter_value currently being edited.

_open_header_editor(*column*)

spinetoolbox.widgets.time_series_fixed_resolution_editor

Contains logic for the fixed step time series editor widget.

Module Contents

Classes

TimeSeriesFixedResolutionEditor	A widget for editing time series data with a fixed time step.

Functions

_resolution_to_text(resolution)	Converts a list of durations into a string of comma-separated durations.
_text_to_resolution(text)	Converts a comma-separated string of durations into a resolution array.

spinetoolbox.widgets.time_series_fixed_resolution_editor.**_resolution_to_text**(*resolution*)

Converts a list of durations into a string of comma-separated durations.

spinetoolbox.widgets.time_series_fixed_resolution_editor.**_text_to_resolution**(*text*)

Converts a comma-separated string of durations into a resolution array.

class spinetoolbox.widgets.time_series_fixed_resolution_editor.**TimeSeriesFixedResolutionEditor**(*parent=No*

Bases: PySide6.QtWidgets.QWidget

A widget for editing time series data with a fixed time step.

Parameters
parent (*QWidget*) – a parent widget

_resolution_changed()

Updates the models after resolution change.

_show_table_context_menu(*position*)

Shows the table's context menu.

Parameters
position (*QPoint*) – menu's position in table view's coordinates

_select_date(*selected_date*)

set_value(*value*)

Sets the parameter_value for editing in this widget.

_show_calendar()

_start_time_changed()

Updates the model due to start time change.

_update_plot(*topLeft=None*, *bottomRight=None*, *roles=None*)

Updated the plot.

value()

Returns the parameter_value currently being edited.

_open_header_editor(*column*)

spinetoolbox.widgets.time_series_variable_resolution_editor

Contains logic for the variable resolution time series editor widget.

Module Contents

Classes

TimeSeriesVariableResolutionEditor	A widget for editing variable resolution time series data.

class spinetoolbox.widgets.time_series_variable_resolution_editor.**TimeSeriesVariableResolutionEditor**(*pa*

Bases: PySide6.QtWidgets.QWidget

A widget for editing variable resolution time series data.

Parameters
parent (*QWidget*) – a parent widget

_show_table_context_menu(*position*)

Shows the table's context menu.

Parameters
position (*QPoint*) – menu's position on the table

set_value(*value*)

Sets the time series being edited.

_update_plot(*topLeft=None*, *bottomRight=None*, *roles=None*)

Updates the plot widget.

value()

Return the time series currently being edited.

_open_header_editor(*column*)

spinetoolbox.widgets.toolbars

Functions to make and handle QToolBars.

Module Contents

Classes

_TitleWidget	

ToolBar	Base class for Toolbox toolbars.
PluginToolBar	A plugin toolbar.
SpecToolBar	Base class for Toolbox toolbars.
ItemsToolBar	The base items
ExecuteToolBar	Base class for Toolbox toolbars.

class spinetoolbox.widgets.toolbars.**_TitleWidget**(*title*, *toolbar*)

Bases: PySide6.QtWidgets.QWidget

sizeHint()

paintEvent(*ev*)

do_paint(*painter*, *x=None*)

class spinetoolbox.widgets.toolbars.**ToolBar**(*name*, *toolbox*)

Bases: PySide6.QtWidgets.QToolBar

Base class for Toolbox toolbars.

Parameters

- **name** (*str*) – toolbar's name
- **toolbox** (ToolboxUI) – Toolbox main window

name()

paintEvent(*ev*)

set_colored_icons(*colored*)

set_color(*color*)

Sets toolbar's background color.

Parameters
color (*QColor*) – background color

set_project_actions_enabled(*enabled*)

Enables or disables project related actions.

Parameters
enabled (*bool*) – True to enable actions, False to disable

_process_tool_button(*button*)

_insert_tool_button(*before*, *button*)

Inserts button into the toolbar.

Parameters

- **before** (*QWidget*) – insert before this widget
- **button** (*QToolButton*) – button to add

Returns
QAction

_add_tool_button(*button*)

Adds a button to the toolbar.

Parameters
button (*QToolButton*) – button to add

Returns
QAction

_make_tool_button(*icon*, *text*, *slot=None*, *tip=None*)

Makes a new tool button and adds it to the toolbar.

Parameters

- **icon** (*QIcon*) – button's icon
- **text** (*str*) – button's text
- **slot** (*Callable*) – slot where to connect button's clicked signal
- **tip** (*str*) – button's tooltip

Returns
created button

Return type
QToolButton

_icon_from_factory(*factory*)

class spinetoolbox.widgets.toolbars.**PluginToolBar**(*name*, *parent*)

Bases: *ToolBar*

A plugin toolbar.

Parameters
parent (ToolboxUI) – QMainWindow instance

name()

buttons()

setup(*plugin_specs*, *disabled_names*)

Sets up the toolbar.

Parameters

- **plugin_specs** (*dict*) – mapping from specification name to specification
- **disabled_names** (*Iterable of str*) – specifications that should be disabled

_update_spec_button_name(*old_name*, *new_name*)

class spinetoolbox.widgets.toolbars.**SpecToolBar**(*parent*)

Bases: *ToolBar*

Base class for Toolbox toolbars.

Parameters

- **name** (*str*) – toolbar's name
- **toolbox** (ToolboxUI) – Toolbox main window

buttons()

_insert_specs(*parent*, *first*, *last*)

_add_spec(*row*)

_remove_specs(*parent*, *first*, *last*)

_remove_spec(*row*)

_reset_specs()

setup()

class spinetoolbox.widgets.toolbars.**ItemsToolBar**(*parent*)

Bases: *ToolBar*

The base items

Parameters

parent (ToolboxUI) – QMainWindow instance

_SEPARATOR = ';;'

buttons()

setup()

add_project_item_buttons()

_add_project_item_button(*item_type*, *factory*)

dragLeaveEvent(*event*)

dragEnterEvent(*event*)

dragMoveEvent(*event*)

dropEvent(*event*)

_update_drop_actions(*event*)

Updates source and target actions for drop operation:

Parameters

event (*QDragMoveEvent*) –

paintEvent(*ev*)

Draw a line as drop indicator.

_drop_line()

icon_ordering()

class spinetoolbox.widgets.toolbars.**ExecuteToolBar**(*parent*)

Bases: *ToolBar*

Base class for Toolbox toolbars.

Parameters
parent (ToolboxUI) – QMainWindow instance

setup()

_add_button_from_action(*action*)

_add_buttons()

Adds buttons to the toolbar.

20.1.2 Submodules

spinetoolbox.__main__

Spine Toolbox application main file.

Module Contents

spinetoolbox.__main__.**return_code**

spinetoolbox._version

Module Contents

spinetoolbox._version.**TYPE_CHECKING = False**

spinetoolbox._version.**VERSION_TUPLE**

spinetoolbox._version.**version: str**

spinetoolbox._version.**__version__: str**

spinetoolbox._version.**__version_tuple__: VERSION_TUPLE**

spinetoolbox._version.**version_tuple: VERSION_TUPLE**

spinetoolbox.config

Application constants and style sheets.

Module Contents

spinetoolbox.config.**LATEST_PROJECT_VERSION** = 13

spinetoolbox.config.**REQUIRED_SPINE_OPT_VERSION** = '0.6.9'

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

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

spinetoolbox.config.**_frozen**

spinetoolbox.config.**_path_to_executable**

spinetoolbox.config.**APPLICATION_PATH**

spinetoolbox.config.**_program_root**

spinetoolbox.config.**DEFAULT_WORK_DIR**

spinetoolbox.config.**DOCUMENTATION_PATH**

spinetoolbox.config.**ONLINE_DOCUMENTATION_URL** = 'https://spine-toolbox.readthedocs.io/en/master/'

spinetoolbox.config.**PLUGINS_PATH**

spinetoolbox.config.**PLUGIN_REGISTRY_URL** = 'https://spine-tools.github.io/PluginRegistry/registry.json'

spinetoolbox.config.**JUPYTER_KERNEL_TIME_TO_DEAD** = 20

spinetoolbox.config.**PROJECT_FILENAME** = 'project.json'

spinetoolbox.config.**PROJECT_LOCAL_DATA_DIR_NAME** = 'local'

spinetoolbox.config.**PROJECT_LOCAL_DATA_FILENAME** = 'project_local_data.json'

spinetoolbox.config.**SPECIFICATION_LOCAL_DATA_FILENAME** = 'specification_local_data.json'

spinetoolbox.config.**PROJECT_ZIP_FILENAME** = 'project_package'

spinetoolbox.config.**STATUSBAR_SS** = 'QStatusBar{background-color: #EBEBE0; border-width: 1px; border-color: gray; border-style: groove;}'

spinetoolbox.config.**BG_COLOR** = '#19232D'

spinetoolbox.config.**FG_COLOR** = '#F0F0F0'

spinetoolbox.config.**SETTINGS_SS**

spinetoolbox.config.**TEXTBROWSER_SS**

spinetoolbox.config.**MAINWINDOW_SS**

spinetoolbox.execution_managers

Classes to manage tool instance execution in various forms.

Module Contents

Classes

ExecutionManager	Base class for all tool instance execution managers.
QProcessExecutionManager	Class to manage tool instance execution using a PySide6 QProcess.

class spinetoolbox.execution_managers.**ExecutionManager**(*logger*)

Bases: PySide6.QtCore.QObject

Base class for all tool instance execution managers.

Class constructor.

Parameters
logger (LoggerInterface) – a logger instance

execution_finished

abstract start_execution(*workdir=None*)

Starts the execution.

Parameters
workdir (*str*) – Work directory

abstract stop_execution()

Stops the execution.

class spinetoolbox.execution_managers.**QProcessExecutionManager**(*logger*, *program=''*, *args=None*, *silent=False*, *semisilent=False*)

Bases: *ExecutionManager*

Class to manage tool instance execution using a PySide6 QProcess.

Class constructor.

Parameters

- **logger** (LoggerInterface) – a logger instance
- **program** (*str*) – Path to program to run in the subprocess (e.g. julia.exe)
- **args** (*list, optional*) – List of argument for the program (e.g. path to script file)
- **silent** (*bool*) – Whether or not to emit logger msg signals
- **semisilent** (*bool*) – If True, show Process Log messages

program()

Program getter method.

args()

Program argument getter method.

start_execution(*workdir=None*)

Starts the execution of a command in a QProcess.

Parameters
workdir (*str, optional*) – Work directory

wait_for_process_finished(*msecs=30000*)

Wait for subprocess to finish.

Parameters
msecs (*int*) – Timeout in milliseconds

Returns
True if process finished successfully, False otherwise

process_started()

Run when subprocess has started.

on_state_changed(*new_state*)

Runs when QProcess state changes.

Parameters
new_state (*int*) – Process state number (QProcess::ProcessState)

on_process_error(*process_error*)

Runs if there is an error in the running QProcess.

Parameters
process_error (*int*) – Process error number (QProcess::ProcessError)

teardown_process()

Tears down the QProcess in case a QProcess.ProcessError occurred. Emits execution_finished signal.

stop_execution()

See base class.

on_process_finished(*exit_code*, *exit_status*)

Runs when subprocess has finished.

Parameters

- **exit_code** (*int*) – Return code from external program (only valid for normal exits)
- **exit_status** (*int*) – Crash or normal exit (QProcess::ExitStatus)

on_ready_stdout()

Emit data from stdout.

on_ready_stderr()

Emit data from stderr.

spinetoolbox.fetch_parent

The FetchParent and FlexibleFetchParent classes.

Module Contents

Classes

FetchParent	**param index** an index to speedup looking up fetched items
ItemTypeFetchParent	**param index** an index to speedup looking up fetched items
FlexibleFetchParent	**param index** an index to speedup looking up fetched items
FetchIndex	dict() -> new empty dictionary
_ItemCallback	

class spinetoolbox.fetch_parent.**FetchParent**(*index=None*, *owner=None*, *chunk_size=1000*)

Bases: PySide6.QtCore.QObject

Parameters

- **index** (FetchIndex, *optional*) – an index to speedup looking up fetched items
- **owner** (*object, optional*) – somebody who owns this FetchParent. If it's a QObject instance, then this FetchParent becomes obsolete whenever the owner is destroyed
- **chunk_size** (*int, optional*) – the number of items this parent should be happy with fetching at a time. If None, then no limit is imposed and the parent should fetch the entire contents of the DB.

property index

abstract property fetch_item_type

Returns the DB item type to fetch, e.g., "entity_class".

Returns

str

property is_obsolete

property is_fetched

property is_busy

_changes_pending

apply_changes_immediately()

reset()

Resets fetch parent as if nothing was ever fetched.

position(*db_map*)

increment_position(*db_map*)

_apply_pending_changes()

bind_item(*item*, *db_map*)

_make_restore_item_callback(*db_map*)

_make_update_item_callback(*db_map*)

_make_remove_item_callback(*db_map*)

_is_valid(*version*)

_change_item(*handler*, *item*, *db_map*, *version*)

add_item(*item*, *db_map*, *version=None*)

update_item(*item*, *db_map*, *version=None*)

remove_item(*item*, *db_map*, *version=None*)

key_for_index(*db_map*)

Returns the key for this parent in the index.

Parameters
db_map (*DatabaseMapping*) –

Returns
Any

accepts_item(*item*, *db_map*)

Called by the associated SpineDBWorker whenever items are fetched and also added/updated/removed. Returns whether this parent accepts that item as a children.

In case of modifications, the SpineDBWorker will call one or more of `handle_items_added()`, `handle_items_updated()`, or `handle_items_removed()` with all the items that pass this test.

Parameters

- **item** (*dict*) – The item
- **db_map** (*DatabaseMapping*) –

Returns
bool

shows_item(*item*, *db_map*)

Called by the associated SpineDBWorker whenever items are fetched and accepted. Returns whether this parent will show this item to the user.

Parameters

- **item** (*dict*) – The item

- **db_map** (*DatabaseMapping*) –

Returns
bool

`set_obsolete`(*obsolete*)

Sets the obsolete status.

Parameters
obsolete (*bool*) – whether parent has become obsolete

`set_fetched`(*fetched*)

Sets the fetched status.

Parameters
fetched (*bool*) – whether parent has been fetched completely

`set_busy`(*busy*)

Sets the busy status.

Parameters
busy (*bool*) – whether parent is busy fetching

`abstract handle_items_added`(*db_map_data*)

Called by SpineDBWorker when items are added to the DB.

Parameters
db_map_data (*dict*) – Mapping DatabaseMapping instances to list of dict-items for which `accepts_item()` returns True.

`abstract handle_items_removed`(*db_map_data*)

Called by SpineDBWorker when items are removed from the DB.

Parameters
db_map_data (*dict*) – Mapping DatabaseMapping instances to list of dict-items for which `accepts_item()` returns True.

`abstract handle_items_updated`(*db_map_data*)

Called by SpineDBWorker when items are updated in the DB.

Parameters
db_map_data (*dict*) – Mapping DatabaseMapping instances to list of dict-items for which `accepts_item()` returns True.

`class` spinetoolbox.fetch_parent.**ItemTypeFetchParent**(*fetch_item_type*, *index=None*, *owner=None*, *chunk_size=1000*)

Bases: *FetchParent*

Parameters

- **index** (*FetchIndex, optional*) – an index to speedup looking up fetched items
- **owner** (*object, optional*) – somebody who owns this FetchParent. If it's a QObject instance, then this FetchParent becomes obsolete whenever the owner is destroyed
- **chunk_size** (*int, optional*) – the number of items this parent should be happy with fetching at a time. If None, then no limit is imposed and the parent should fetch the entire contents of the DB.

`property fetch_item_type`

Returns the DB item type to fetch, e.g., "entity_class".

Returns
str

abstract handle_items_added(*db_map_data*)

Called by SpineDBWorker when items are added to the DB.

Parameters
db_map_data (*dict*) – Mapping DatabaseMapping instances to list of dict-items for which `accepts_item()` returns True.

abstract handle_items_removed(*db_map_data*)

Called by SpineDBWorker when items are removed from the DB.

Parameters
db_map_data (*dict*) – Mapping DatabaseMapping instances to list of dict-items for which `accepts_item()` returns True.

abstract handle_items_updated(*db_map_data*)

Called by SpineDBWorker when items are updated in the DB.

Parameters
db_map_data (*dict*) – Mapping DatabaseMapping instances to list of dict-items for which `accepts_item()` returns True.

__str__()

class spinetoolbox.fetch_parent.**FlexibleFetchParent**(*fetch_item_type*, *handle_items_added=None*, *handle_items_removed=None*, *handle_items_updated=None*, *accepts_item=None*, *shows_item=None*, *key_for_index=None*, *index=None*, *owner=None*, *chunk_size=1000*)

Bases: *ItemTypeFetchParent*

Parameters

- **index** (*FetchIndex, optional*) – an index to speedup looking up fetched items
- **owner** (*object, optional*) – somebody who owns this FetchParent. If it's a QObject instance, then this FetchParent becomes obsolete whenever the owner is destroyed
- **chunk_size** (*int, optional*) – the number of items this parent should be happy with fetching at a time. If None, then no limit is imposed and the parent should fetch the entire contents of the DB.

key_for_index(*db_map*)

Returns the key for this parent in the index.

Parameters
db_map (*DatabaseMapping*) –

Returns
Any

handle_items_added(*db_map_data*)

Called by SpineDBWorker when items are added to the DB.

Parameters
db_map_data (*dict*) – Mapping DatabaseMapping instances to list of dict-items for which `accepts_item()` returns True.

handle_items_removed(*db_map_data*)

Called by SpineDBWorker when items are removed from the DB.

Parameters
db_map_data (*dict*) – Mapping DatabaseMapping instances to list of dict-items for which accepts_item() returns True.

handle_items_updated(*db_map_data*)

Called by SpineDBWorker when items are updated in the DB.

Parameters
db_map_data (*dict*) – Mapping DatabaseMapping instances to list of dict-items for which accepts_item() returns True.

accepts_item(*item*, *db_map*)

Called by the associated SpineDBWorker whenever items are fetched and also added/updated/removed. Returns whether this parent accepts that item as a children.

In case of modifications, the SpineDBWorker will call one or more of handle_items_added(), handle_items_updated(), or handle_items_removed() with all the items that pass this test.

Parameters

- **item** (*dict*) – The item
- **db_map** (*DatabaseMapping*) –

Returns
bool

shows_item(*item*, *db_map*)

Called by the associated SpineDBWorker whenever items are fetched and accepted. Returns whether this parent will show this item to the user.

Parameters

- **item** (*dict*) – The item
- **db_map** (*DatabaseMapping*) –

Returns
bool

class spinetoolbox.fetch_parent.**FetchIndex**

Bases: dict

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's

(key, value) pairs

dict(iterable) -> new dictionary initialized as if via:
d = {} for k, v in iterable:

d[k] = v

dict(kwargs) -> new dictionary initialized with the name=value pairs**
in the keyword argument list. For example: dict(one=1, two=2)

Initialize self. See help(type(self)) for accurate signature.

reset()

abstract process_item(*item*, *db_map*)

position(*db_map*)

increment_position(*db_map*)

get_items(*key*, *db_map*)

class spinetoolbox.fetch_parent.**_ItemCallback**(*fn*, **args*)

__call__(*item*)

__str__()

Return str(self).

spinetoolbox.headless

Contains facilities to open and execute projects without GUI.

Module Contents

Classes

HeadlessLogger	A LoggerInterface compliant logger that uses Python's standard logging facilities.
ModifiableProject	A simple project that is available for modification script.
ActionsWithProject	A 'task' which opens Toolbox project and operates on it.
Status	Status codes returned from headless execution.

Functions

headless_main(args)	Executes a project using QCoreApplication.
open_project(project_dict, project_dir, logger)	Opens a project.
_specification_dicts(project_dict, project_dir, logger)	Loads project item specification dictionaries.
solve_project_dir(pd)	Makes given path object OS independent.

class spinetoolbox.headless.**HeadlessLogger**

Bases: PySide6.QtCore.QObject

A LoggerInterface compliant logger that uses Python's standard logging facilities.

msg

Emits a notification message.

msg_success

Emits a message on success

msg_warning

Emits a warning message.

msg_error

Emits an error message.

msg_proc

Emits a message originating from a subprocess (usually something printed to stdout).

msg_proc_error

Emits an error message originating from a subprocess (usually something printed to stderr).

information_box

Requests an 'information message box' (e.g. a message window) to be opened with a given title and message.

error_box

Requests an 'error message box' to be opened with a given title and message.

_log_message(*message*)

Prints an information message.

_log_warning(*message*)

Prints a warning message.

_log_error(*message*)

Prints an error message.

_show_information_box(*title*, *message*)

Prints an information message with a title.

_show_error_box(*title*, *message*)

Prints an error message with a title.

_print(*message*, *out_stream*)

Filters HTML tags from message before printing it to given file.

class spinetoolbox.headless.**ModifiableProject**(*project_dir*, *items_dict*, *connection_dicts*)

A simple project that is available for modification script.

Parameters

- **project_dir** (*Path*) – project directory
- **items_dict** (*dict*) – project item dictionaries
- **connection_dicts** (*list of dict*) – connection dictionaries

property project_dir

find_connection(*source_name*, *destination_name*)

Searches for a connection between given items.

Parameters

- **source_name** (*str*) – source item's name
- **destination_name** (*str*) – destination item's name

Returns

connection instance or None if there is no connection

Return type

Connection

find_item(*name*)

Searches for a project item.

Parameters
name (*str*) – item's name

Returns
item dict or None if no such item exists

Return type
dict

items_to_dict()

Stores project items back to dictionaries.

Returns
item dictionaries

Return type
dict

connections_to_dict()

Stores connections back to dictionaries.

Returns
connection dictionaries

Return type
list of dict

class spinetoolbox.headless.**ActionsWithProject**(*args*, *startup_event_type*, *parent*)

Bases: PySide6.QtCore.QObject

A 'task' which opens Toolbox project and operates on it.

The execution of this task is triggered by sending it a 'startup' QEvent using e.g. QCoreApplication.postEvent()

Parameters

- **args** (*argparse.Namespace*) – parsed command line arguments
- **startup_event_type** (*int*) – expected type id for the event that starts this task
- **parent** (*QObject*) – a parent object

_start

A private signal to actually start execution. Not to be used directly. Post a startup event instead.

_dags()

_execute()

Executes this task.

_open_project()

Opens a project.

Returns
status code

Return type
Status

_check_project_version(*project_dict*)

Checks project dict version.

Parameters
project_dict (*dict*) – project dict

Returns
status code

Return type
Status

_exec_mod_script()

Executes project modification script given in command line arguments.

Returns
status code

Return type
Status

_execute_project()

Executes all DAGs in a project.

Returns
status code

Return type
Status

_process_engine_event(*event_type*, *data*)

event(*e*)

_handle_node_execution_started(*data*)

Starts collecting messages from given node.

Parameters
data (*dict*) – execution start data

_handle_node_execution_finished(*data*)

Prints messages for finished nodes.

Parameters
data (*dict*) – execution end data

_handle_event_msg(*data*)

Stores event messages for later printing.

Parameters
data (*dict*) – event message data

_handle_process_msg(*data*)

Stores process messages for later printing.

Parameters
data (*dict*) – process message data

_handle_standard_execution_msg(*data*)

Handles standard execution messages.

Currently, these messages are ignored.

Parameters
data (*dict*) – execution message data

_handle_persistent_execution_msg(*data*)

Handles persistent execution messages.

Parameters
data (*dict*) – execution message data

_handle_kernel_execution_msg(*data*)

Handles kernel messages.

Currently, these messages are ignored.

Parameters
data (*dict*) – message data

_handle_server_status_msg(*data*)

Handles received remote execution messages.

_read_server_config()

Reads the user provided server settings file that the client requires to establish connection.

Returns
Dictionary containing the EngineClient settings or None if the given config file does not exist.

Return type
dict

_insert_remote_engine_settings(*settings*)

Inserts remote engine client settings into the settings dictionary that is delivered to the engine.

Parameters
settings (*dict*) – Original settings dictionary

Returns
Settings dictionary containing remote engine client settings

Return type
dict

_prepare_remote_execution()

If remote execution is enabled, makes an EngineClient for pinging and uploading the project. If ping is successful, the project is uploaded to the server. If the upload is successful, the server responds with a Job id, which is later used by the client to make a 'start execution' request.

Returns

Job id if server is ready for remote execution, empty string if something went wrong
or "1" if local execution is enabled.

Return type
str

spinetoolbox.headless.**headless_main**(*args*)

Executes a project using QCoreApplication.

Parameters
args (*argparser.Namespace*) – parsed command line arguments.

Returns
exit status code; 0 for success, everything else for failure

Return type
int

spinetoolbox.headless.**open_project**(*project_dict*, *project_dir*, *logger*)

Opens a project.

Parameters

- **project_dict** (*dict*) – a serialized project dictionary
- **project_dir** (*Path*) – path to a directory containing the .spinetoolbox dir
- **logger** (LoggerInterface) – a logger

Returns
item dicts, specification dicts, connection dicts, jump dicts and a DagHandler object

Return type
tuple

spinetoolbox.headless.**_specification_dicts**(*project_dict*, *project_dir*, *logger*)

Loads project item specification dictionaries.

Parameters

- **project_dict** (*dict*) – a serialized project dictionary
- **project_dir** (*str*) – path to a directory containing the .spinetoolbox dir
- **logger** (LoggerInterface) – a logger

Returns
a mapping from item type to a list of specification dicts

Return type
dict

spinetoolbox.headless.**solve_project_dir**(*pd*)

Makes given path object OS independent.

Parameters
pd (*Path*) – Path Object

Returns
OS independent path as string.

Return type
str

class spinetoolbox.headless.**Status**

Bases: enum.IntEnum

Status codes returned from headless execution.

Initialize self. See help(type(self)) for accurate signature.

OK = 0

ERROR = 1

ARGUMENT_ERROR = 2

spinetoolbox.helpers

General helper functions and classes.

Module Contents

Classes

LinkType	Graphics scene's link types.
IconListManager	A class to manage icons for icon list widgets.
TransparentIconEngine	Specialization of QIconEngine with transparent background.
CharIconEngine	Specialization of QIconEngine used to draw font-based icons.
ColoredIcon	
ColoredIconEngine	
ProjectDirectoryIconProvider	QFileIconProvider that provides a Spine icon to the
ChildCyclingKeyPressFilter	Event filter class for catching next and previous child key presses.
QuietLogger	
SignalWaiter	A 'traffic light' that allows waiting for a signal to be emitted in another thread.
CustomSyntaxHighlighter	
HTMLTagFilter	HTML tag filter.
SealCommand	A 'meta' command that does not store undo data but can be used in mergeWith methods of other commands.

Functions

home_dir()	Returns user's home dir
format_log_message(msg_type, message[, show_datetime])	Adds color tags and optional time stamp to message.
busy_effect(func)	Decorator to change the mouse cursor to 'busy' while a function is processed.
create_dir(base_path[, folder, verbosity])	Create (input/output) directories recursively.
rename_dir(old_dir, new_dir, toolbox, box_title)	Renames directory.
open_url(url)	Opens the given url in the appropriate Web browser for the user's desktop environment,
set_taskbar_icon()	Set application icon to Windows taskbar.
supported_img_formats()	Checks if reading .ico files is supported.
pyside6_version_check()	Check that PySide6 version is at least 6.4.
get_datetime(show[, date])	Returns date and time string for appending into Event Log messages.
copy_files(src_dir, dst_dir[, includes, excludes])	Function for copying files. Does not copy folders.

continues on next page

Table 1 – continued from previous page

erase_dir(path[, verbosity])	Deletes a directory and all its contents without prompt.
recursive_overwrite(logger, src, dst[, ignore, silent])	Copies everything from source directory to destination directory recursively.
tuple_itemgetter(itemgetter_func, num_indexes)	Change output of itemgetter to always be a tuple even for a single index.
format_string_list(str_list)	Returns a html unordered list from the given list of strings.
rows_to_row_count_tuples(rows)	Breaks a list of rows into a list of (row, count) tuples corresponding to chunks of successive rows.
object_icon(display_icon)	Creates and returns a QIcon corresponding to display_icon.
color_pixmap(pixmap, color)	
make_icon_id(icon_code, color_code)	Takes icon and color codes, and return equivalent integer.
interpret_icon_id(display_icon)	Takes a display icon id and returns an equivalent tuple of icon and color code.
default_icon_id()	Creates a default icon id.
ensure_window_is_on_screen(window, size)	Checks if window is on screen and if not, moves and resizes it to make it visible on the primary screen.
first_non_null(s)	Returns the first element in Iterable s that is not None.
get_save_file_name_in_last_dir(qsettings, key, parent, ...)	Calls QFileDialog.getSaveFileName in the directory that was selected last time the dialog was accepted.
get_open_file_name_in_last_dir(qsettings, key, parent, ...)	
try_number_from_string(text)	Tries to convert a string to integer or float.
focused_widget_has_callable(parent, callable_name)	Returns True if the currently focused widget or one of its ancestors has the given callable.
call_on_focused_widget(parent, callable_name)	Calls the given callable on the currently focused widget or one of its ancestors.
select_gams_executable(parent, line_edit)	Opens file browser where user can select a Gams executable (i.e. gams.exe on Windows).
select_julia_executable(parent, line_edit)	Opens file browser where user can select a Julia executable (i.e. julia.exe on Windows).
select_julia_project(parent, line_edit)	Shows file browser and inserts selected julia project dir to give line_edit.
select_python_interpreter(parent, line_edit)	Opens file browser where user can select a python interpreter (i.e. python.exe on Windows).
select_conda_executable(parent, line_edit)	Opens file browser where user can select a conda executable.
is_valid_conda_executable(p)	Checks that given path points to an existing file and the file name starts with 'conda'.
select_certificate_directory(parent, line_edit)	Shows file browser and inserts selected certificate directory to given line edit.
file_is_valid(parent, file_path, msgbox_title[, ...])	Checks that given path is not a directory and it's a file that actually exists.
dir_is_valid(parent, dir_path, msgbox_title)	Checks that given path is a directory. Needed in
make_settings_dict_for_engine(app_settings)	Converts Toolbox settings to a dictionary acceptable by Engine.
make_icon_background(color)	

continues on next page

Table 1 – continued from previous page

make_icon_toolbar_ss(color)	
color_from_index(i, count[, base_hue, saturation, value])	
unique_name(prefix, existing)	Creates a unique name in the form *prefix (xx)* where xx is a counter value.
parse_specification_file(spec_path, logger)	Parses specification file.
load_specification_from_file(spec_path, ...)	Returns an Item specification from a definition file.
specification_from_dict(spec_dict, local_data_dict, ...)	Returns item specification from a dictionary.
plugins_dirs(app_settings)	Loads plugins.
load_plugin_dict(plugin_dir, logger)	Loads plugin dict from plugin directory.
load_plugin_specifications(plugin_dict, ...)	Loads plugin's specifications.
load_specification_local_data(config_dir)	Loads specifications' project-specific data.
parameter_identifier(database, parameter, names, ...)	Concatenates given information into parameter value identifier string.
disconnect(signal, *slots)	Disconnects signal for the duration of a 'with' block.
signal_waiter(signal[, condition, timeout])	Gives a context manager that waits for the emission of given Qt signal.
inquire_index_name(model, column, title, parent_widget)	Asks for indexed parameter's index name and updates model accordingly.
preferred_row_height(widget[, factor])	
restore_ui(window, app_settings, settings_group)	Restores UI state from previous session.
save_ui(window, app_settings, settings_group)	Saves UI state for next session.
bisect_chunks(current_data, new_data[, key])	Finds insertion points for chunks of data using binary search.
load_project_dict(project_config_dir, logger)	Loads project dictionary from project directory.
load_local_project_data(project_config_dir, logger)	Loads local project data.
merge_dicts(source, target)	Merges two dictionaries that may contain nested dictionaries recursively.
fix_lightness_color(color[, lightness])	
scrolling_to_bottom(widget[, tolerance])	
_is_metadata_item(item)	Identifies a database metadata record.
same_path(path1, path2)	Checks if two paths are equal.
solve_connection_file(connection_file, ...)	Returns the connection_file path, if it exists on this computer. If the path
remove_first(lst, items)	
plain_to_rich(text)	Turns plain strings into rich text.
list_to_rich_text(data)	Turns a sequence of strings into rich text list.
plain_to_tool_tip(text)	Turns plain strings into rich text and empty strings/Nones to None.

Attributes

_matplotlib_version	
DB_ITEM_SEPARATOR	Display string to separate items such as entity names.

spinetoolbox.helpers.**_matplotlib_version**

class spinetoolbox.helpers.**LinkType**

Bases: enum.Enum

Graphics scene's link types.

CONNECTION = 'connection'

JUMP = 'jump'

spinetoolbox.helpers.**home_dir**()

Returns user's home dir

spinetoolbox.helpers.**format_log_message**(*msg_type*, *message*, *show_datetime=True*)

Adds color tags and optional time stamp to message.

Parameters

- **msg_type** (*str*) – message's type; accepts only 'msg', 'msg_success', 'msg_warning', or 'msg_error'
- **message** (*str*) – message to format
- **show_datetime** (*bool*) – True to add time stamp, False to omit it

Returns
formatted message

Return type
str

spinetoolbox.helpers.**busy_effect**(*func*)

Decorator to change the mouse cursor to 'busy' while a function is processed.

Parameters
func (*Callable*) – Decorated function.

spinetoolbox.helpers.**create_dir**(*base_path*, *folder=''*, *verbosity=False*)

Create (input/output) directories recursively.

Parameters

- **base_path** (*str*) – Absolute path to wanted dir
- **folder** (*str*) – (Optional) Folder name. Usually short name of item.
- **verbosity** (*bool*) – True prints a message that tells if the directory already existed or if it was created.

Raises
OSError if operation failed. –

spinetoolbox.helpers.**rename_dir**(*old_dir*, *new_dir*, *toolbox*, *box_title*)

Renames directory.

Parameters

- **old_dir** (*str*) – Absolute path to directory that will be renamed
- **new_dir** (*str*) – Absolute path to new directory
- **toolbox** (ToolboxUI) – A toolbox to log messages and ask questions.
- **box_title** (*str*) – The title of the message boxes, (e.g. "Undoing 'rename DC1 to DC2'")

Returns
True if operation was successful, False otherwise

Return type
bool

spinetoolbox.helpers.**open_url**(*url*)

Opens the given url in the appropriate Web browser for the user's desktop environment, and returns true if successful; otherwise returns false.

If the URL is a reference to a local file (i.e., the URL scheme is "file") then it will be opened with a suitable application instead of a Web browser.

Handle return value on caller side.

Parameters
url (*str*) – URL to open

Returns
True if successful, False otherwise

Return type
bool

spinetoolbox.helpers.**set_taskbar_icon**()

Set application icon to Windows taskbar.

spinetoolbox.helpers.**supported_img_formats**()

Checks if reading .ico files is supported.

spinetoolbox.helpers.**pyside6_version_check**()

Check that PySide6 version is at least 6.4.

qt_version (str) is the Qt version used to compile PySide6. E.g. "6.4.1" qt_version_info (tuple) contains each version component separately e.g. (6, 4, 1)

spinetoolbox.helpers.**get_datetime**(*show*, *date=True*)

Returns date and time string for appending into Event Log messages.

Parameters

- **show** (*bool*) – True returns date and time string. False returns empty string.
- **date** (*bool*) – Whether or not the date should be included in the result

Returns
datetime string or empty string if show is False

Return type
str

spinetoolbox.helpers.**copy_files**(*src_dir*, *dst_dir*, *includes=None*, *excludes=None*)

Function for copying files. Does not copy folders.

Parameters

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

Returns
Number of files copied

Return type
count (int)

spinetoolbox.helpers.**erase_dir**(*path*, *verbosity=False*)

Deletes a directory and all its contents without prompt.

Parameters

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

Returns
True if operation was successful, False otherwise

Return type
bool

spinetoolbox.helpers.**recursive_overwrite**(*logger*, *src*, *dst*, *ignore=None*, *silent=True*)

Copies everything from source directory to destination directory recursively. Overwrites existing files.

Parameters

- **logger** (LoggerInterface) – Enables e.g. printing to Event Log
- **src** (*str*) – Source directory
- **dst** (*str*) – Destination directory
- **ignore** (*Callable, optional*) – Ignore function
- **silent** (*bool*) – If False, messages are sent to Event Log, If True, copying is done in silence

spinetoolbox.helpers.**tuple_itemgetter**(*itemgetter_func*, *num_indexes*)

Change output of itemgetter to always be a tuple even for a single index.

Parameters

- **itemgetter_func** (*Callable*) – item getter function
- **num_indexes** (*int*) – number of indexes

Returns
getter function that works with a single index

Return type
Callable

spinetoolbox.helpers.**format_string_list**(*str_list*)

Returns a html unordered list from the given list of strings. Intended to print error logs as returned by spinedb_api.

Parameters
str_list (*list of str*) – list of strings to format

Returns
formatted list

Return type
str

spinetoolbox.helpers.**rows_to_row_count_tuples**(*rows*)

Breaks a list of rows into a list of (row, count) tuples corresponding to chunks of successive rows.

Parameters
rows (*Iterable of int*) – rows

Returns
row count tuples

Return type
list of tuple

class spinetoolbox.helpers.**IconListManager**(*icon_size*)

A class to manage icons for icon list widgets.

Parameters
icon_size (*QSize*) – icon's size

init_model()

Init model that can be used to display all icons in a list.

_model_data(*index*, *role*)

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

Parameters

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

Returns
role-dependent model data

Return type
Any

spinetoolbox.helpers.**object_icon**(*display_icon*)

Creates and returns a QIcon corresponding to display_icon.

Parameters
display_icon (*int*) – icon id

Returns
requested icon

Return type
QIcon

class spinetoolbox.helpers.**TransparentIconEngine**

Bases: PySide6.QtGui.QIconEngine

Specialization of QIconEngine with transparent background.

pixmap(*size=QSize(512, 512)*, *mode=None*, *state=None*)

class spinetoolbox.helpers.**CharIconEngine**(*char*, *color=None*)

Bases: *TransparentIconEngine*

Specialization of QIconEngine used to draw font-based icons.

Parameters

- **char** (*str*) – character to use as the icon
- **color** (*QColor, optional*) –

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

class spinetoolbox.helpers.**ColoredIcon**(*icon_file_name*, *icon_color*, *icon_size*, *colored=None*)

Bases: PySide6.QtGui.QIcon

set_colored(*colored*)

color(*mode=QIcon.Normal*)

class spinetoolbox.helpers.**ColoredIconEngine**(*icon_file_name*, *icon_color*, *icon_size*, *colored=None*)

Bases: PySide6.QtGui.QIconEngine

color(*mode=QIcon.Normal*)

set_colored(*colored*)

_do_make_pixmap(*mode*, *state*)

_make_pixmap(*mode*, *state*)

pixmap(*size*, *mode*, *state*)

spinetoolbox.helpers.**color_pixmap**(*pixmap*, *color*)

spinetoolbox.helpers.**make_icon_id**(*icon_code*, *color_code*)

Takes icon and color codes, and return equivalent integer.

Parameters

- **icon_code** (*int*) – icon's code
- **color_code** (*int*) – color code

Returns
icon id

Return type
int

spinetoolbox.helpers.**interpret_icon_id**(*display_icon*)

Takes a display icon id and returns an equivalent tuple of icon and color code.

Parameters
display_icon (*int, optional*) – icon id

Returns
icon's code, color code

Return type
tuple

spinetoolbox.helpers.**default_icon_id**()

Creates a default icon id.

Returns
default icon's id

Return type
int

class spinetoolbox.helpers.**ProjectDirectoryIconProvider**

Bases: PySide6.QtWidgets.QFileIconProvider

QFileIconProvider that provides a Spine icon to the Open Project Dialog when a Spine Toolbox project directory is encountered.

icon(*info*)

Returns an icon for the file described by info.

Parameters
info (*QFileInfo*) – File (or directory) info

Returns
Icon for a file system resource with the given info

Return type
QIcon

spinetoolbox.helpers.**ensure_window_is_on_screen**(*window*, *size*)

Checks if window is on screen and if not, moves and resizes it to make it visible on the primary screen.

Parameters

- **window** (*QWidget*) – a window to check
- **size** (*QSize*) – desired window size if the window is moved

spinetoolbox.helpers.**first_non_null**(*s*)

Returns the first element in Iterable s that is not None.

spinetoolbox.helpers.**get_save_file_name_in_last_dir**(*qsettings*, *key*, *parent*, *caption*, *given_dir*, *filter_=''*)

Calls QFileDialog.getSaveFileName in the directory that was selected last time the dialog was accepted.

Parameters

- **qsettings** (*QSettings*) – A QSettings object where the last directory is stored
- **key** (*string*) – The name of the entry in the above QSettings
- **parent** – Args passed to QFileDialog.getSaveFileName
- **caption** – Args passed to QFileDialog.getSaveFileName
- **given_dir** – Args passed to QFileDialog.getSaveFileName
- **filter** – Args passed to QFileDialog.getSaveFileName

Returns
filename str: selected filter

Return type
str

spinetoolbox.helpers.**get_open_file_name_in_last_dir**(*qsettings*, *key*, *parent*, *caption*, *given_dir*, *filter_=''*)

spinetoolbox.helpers.**try_number_from_string**(*text*)

Tries to convert a string to integer or float.

Parameters
text (*str*) – string to convert

Returns
converted value or text if conversion failed

Return type
int or float or str

spinetoolbox.helpers.**focused_widget_has_callable**(*parent*, *callable_name*)

Returns True if the currently focused widget or one of its ancestors has the given callable.

spinetoolbox.helpers.**call_on_focused_widget**(*parent*, *callable_name*)

Calls the given callable on the currently focused widget or one of its ancestors.

class spinetoolbox.helpers.**ChildCyclingKeyPressFilter**

Bases: PySide6.QtCore.QObject

Event filter class for catching next and previous child key presses. Used in filtering the Ctrl+Tab and Ctrl+Shift+Tab key presses in the Item Properties tab widget.

eventFilter(*obj*, *event*)

spinetoolbox.helpers.**select_gams_executable**(*parent*, *line_edit*)

Opens file browser where user can select a Gams executable (i.e. gams.exe on Windows).

Parameters

- **parent** (*QWidget, optional*) – Parent widget for the file dialog and message boxes
- **line_edit** (*QLineEdit*) – Line edit where the selected path will be inserted

spinetoolbox.helpers.**select_julia_executable**(*parent*, *line_edit*)

Opens file browser where user can select a Julia executable (i.e. julia.exe on Windows). Used in SettingsWidget and KernelEditor.

Parameters

- **parent** (*QWidget, optional*) – Parent widget for the file dialog and message boxes
- **line_edit** (*QLineEdit*) – Line edit where the selected path will be inserted

spinetoolbox.helpers.**select_julia_project**(*parent*, *line_edit*)

Shows file browser and inserts selected julia project dir to give line_edit. Used in SettingsWidget and KernelEditor.

Parameters

- **parent** (*QWidget, optional*) – Parent of QFileDialog
- **line_edit** (*QLineEdit*) – Line edit where the selected path will be inserted

spinetoolbox.helpers.**select_python_interpreter**(*parent*, *line_edit*)

Opens file browser where user can select a python interpreter (i.e. python.exe on Windows). Used in SettingsWidget and KernelEditor.

Parameters

- **parent** (*QWidget*) – Parent widget for the file dialog and message boxes
- **line_edit** (*QLineEdit*) – Line edit where the selected path will be inserted

spinetoolbox.helpers.**select_conda_executable**(*parent*, *line_edit*)

Opens file browser where user can select a conda executable.

Parameters

- **parent** (*QWidget*) – Parent widget for the file dialog and message boxes
- **line_edit** (*QLineEdit*) – Line edit where the selected path will be inserted

spinetoolbox.helpers.**is_valid_conda_executable**(*p*)

Checks that given path points to an existing file and the file name starts with 'conda'.

Parameters

p (*str*) – Absolute path to a file

spinetoolbox.helpers.**select_certificate_directory**(*parent*, *line_edit*)

Shows file browser and inserts selected certificate directory to given line edit.

Parameters

- **parent** (*QWidget, optional*) – Parent of QFileDialog
- **line_edit** (*QLineEdit*) – Line edit where the selected dir path will be inserted

spinetoolbox.helpers.**file_is_valid**(*parent*, *file_path*, *msgbox_title*, *extra_check=None*)

Checks that given path is not a directory and it's a file that actually exists. In addition, can be used to check if the file name in given file path starts with the given extra_check string. Needed in SettingsWidget and KernelEditor because the QLineEdits are editable. Returns True when file_path is an empty string so that we can use default values (e.g. from line edit place holder text). Returns also True when file_path is just 'python' or 'julia' so that user's can use the python or julia in PATH.

Parameters

- **parent** (*QWidget*) – Parent widget for the message boxes
- **file_path** (*str*) – Path to check
- **msgbox_title** (*str*) – Title for message boxes
- **extra_check** (*str, optional*) – String that must match the file name of the given file_path (without extension)

Returns

True if given path is an empty string or if path is valid, False otherwise

Return type

bool

spinetoolbox.helpers.**dir_is_valid**(*parent*, *dir_path*, *msgbox_title*)

Checks that given path is a directory. Needed in SettingsWdiget and KernelEditor because the QLineEdits are editable. Returns True when dir_path is an empty string so that we can use default values (e.g. from line edit place holder text)

Parameters

- **parent** (*QWidget*) – Parent widget for the message box
- **dir_path** (*str*) – Directory path to check
- **msgbox_title** (*str*) – Message box title

Returns
True if given path is an empty string or if path is an existing directory, False otherwise

Return type
bool

class spinetoolbox.helpers.**QuietLogger**

__getattr__(_)

__call__(**args*, ***kwargs*)

spinetoolbox.helpers.**make_settings_dict_for_engine**(*app_settings*)

Converts Toolbox settings to a dictionary acceptable by Engine.

Parameters
app_settings (*QSettings*) – Toolbox settings

Returns
Engine-compatible settings

Return type
dict

spinetoolbox.helpers.**make_icon_background**(*color*)

spinetoolbox.helpers.**make_icon_toolbar_ss**(*color*)

spinetoolbox.helpers.**color_from_index**(*i*, *count*, *base_hue=0.0*, *saturation=1.0*, *value=1.0*)

spinetoolbox.helpers.**unique_name**(*prefix*, *existing*)

Creates a unique name in the form *prefix (xx)* where xx is a counter value. When *prefix* already contains a counter *(xx)*, the value *xx* is updated.

Parameters

- **prefix** (*str*) – name prefix
- **existing** (*Iterable of str*) – existing names

Returns
unique name

Return type
str

spinetoolbox.helpers.**parse_specification_file**(*spec_path*, *logger*)

Parses specification file.

Parameters

- **spec_path** (*str*) – path to specification file
- **logger** (LoggerInterface) – a logger

Returns
specification dict or None if the operation failed

Return type
dict

spinetoolbox.helpers.**load_specification_from_file**(*spec_path*, *local_data_dict*, *spec_factories*, *app_settings*, *logger*)

Returns an Item specification from a definition file.

Parameters

- **spec_path** (*str*) – Path of the specification definition file
- **local_data_dict** (*dict*) – specifications local data dict
- **spec_factories** (*dict*) – Dictionary mapping specification type to ProjectItemSpecificationFactory
- **app_settings** (*QSettings*) – Toolbox settings
- **logger** (LoggerInterface) – a logger

Returns
item specification or None if reading the file failed

Return type
ProjectItemSpecification

spinetoolbox.helpers.**specification_from_dict**(*spec_dict*, *local_data_dict*, *spec_factories*, *app_settings*, *logger*)

Returns item specification from a dictionary.

Parameters

- **spec_dict** (*dict*) – Dictionary with the specification
- **local_data_dict** (*dict*) – specifications local data
- **spec_factories** (*dict*) – Dictionary mapping specification name to ProjectItemSpecificationFactory
- **app_settings** (*QSettings*) – Toolbox settings
- **logger** (LoggerInterface) – a logger

Returns
specification or None if factory isn't found.

Return type
ProjectItemSpecification or NoneType

spinetoolbox.helpers.**plugins_dirs**(*app_settings*)

Loads plugins.

Parameters
app_settings (*QSettings*) – Toolbox settings

Returns
plugin directories

Return type
list of str

spinetoolbox.helpers.**load_plugin_dict**(*plugin_dir*, *logger*)

Loads plugin dict from plugin directory.

Parameters

- **plugin_dir** (*str*) – path of plugin dir with "plugin.json" in it
- **logger** (LoggerInterface) – a logger

Returns
plugin dict or None if the operation failed

Return type
dict

spinetoolbox.helpers.**load_plugin_specifications**(*plugin_dict*, *local_data_dict*, *spec_factories*, *app_settings*, *logger*)

Loads plugin's specifications.

Parameters

- **plugin_dict** (*dict*) – plugin dict
- **local_data_dict** (*dict*) – specifications local data dictionary
- **spec_factories** (*dict*) – Dictionary mapping specification name to ProjectItemSpecificationFactory
- **app_settings** (*QSettings*) – Toolbox settings
- **logger** (LoggerInterface) – a logger

Returns
mapping from plugin name to list of specifications or None if the operation failed

Return type
dict

spinetoolbox.helpers.**load_specification_local_data**(*config_dir*)

Loads specifications' project-specific data.

Parameters
config_dir (*str or Path*) – project config dir

Returns
specifications local data

Return type
dict

spinetoolbox.helpers.**DB_ITEM_SEPARATOR = ' '**

Display string to separate items such as entity names.

spinetoolbox.helpers.**parameter_identifier**(*database*, *parameter*, *names*, *alternative*)

Concatenates given information into parameter value identifier string.

Parameters

- **database** (*str, optional*) – database's code name
- **parameter** (*str*) – parameter's name
- **names** (*list of str*) – name of the entity or class that holds the value
- **alternative** (*str or NoneType*) – name of the value's alternative

spinetoolbox.helpers.**disconnect**(*signal*, **slots*)

Disconnects signal for the duration of a 'with' block.

Parameters

- **signal** (*Signal*) – signal to disconnect
- ***slots** – slots to disconnect from

class spinetoolbox.helpers.**SignalWaiter**(*condition=None*, *timeout=None*)

Bases: PySide6.QtCore.QObject

A 'traffic light' that allows waiting for a signal to be emitted in another thread.

Parameters

- **condition** (*function, optional*) – receiving the self.args and returning whether to stop waiting.
- **timeout** (*float, optional*) – timeout in seconds; wait will raise after timeout

trigger(**args*)

Signal receiving slot.

wait()

Wait for signal to be received.

spinetoolbox.helpers.**signal_waiter**(*signal*, *condition=None*, *timeout=None*)

Gives a context manager that waits for the emission of given Qt signal.

Parameters

- **signal** (*Any*) – signal to wait
- **condition** (*Callable, optional*) – a callable that takes the signal's parameters and returns True to stop waiting
- **timeout** (*float, optional*) – timeout in seconds; if None, wait indefinitely

Yields

SignalWaiter – waiter instance

class spinetoolbox.helpers.**CustomSyntaxHighlighter**(**arg*, ***kwargs*)

Bases: PySide6.QtGui.QSyntaxHighlighter

property formats

set_style(*style*)

yield_formats(*text*)

highlightBlock(*text*)

spinetoolbox.helpers.**inquire_index_name**(*model*, *column*, *title*, *parent_widget*)

Asks for indexed parameter's index name and updates model accordingly.

Parameters

- **model** (IndexedValueTableModel *or* ArrayModel) – a model with header that contains index names
- **column** (*int*) – column index
- **title** (*str*) – input dialog's title
- **parent_widget** (*QWidget*) – dialog's parent widget

spinetoolbox.helpers.**preferred_row_height**(*widget*, *factor=1.5*)

spinetoolbox.helpers.**restore_ui**(*window*, *app_settings*, *settings_group*)

Restores UI state from previous session.

Parameters

- **window** (*QMainWindow*) –
- **app_settings** (*QSettings*) –
- **settings_group** (*str*) –

spinetoolbox.helpers.**save_ui**(*window*, *app_settings*, *settings_group*)

Saves UI state for next session.

Parameters

- **window** (*QMainWindow*) –
- **app_settings** (*QSettings*) –
- **settings_group** (*str*) –

spinetoolbox.helpers.**bisect_chunks**(*current_data*, *new_data*, *key=None*)

Finds insertion points for chunks of data using binary search.

Parameters

- **current_data** (*list*) – sorted list where to insert new data
- **new_data** (*list*) – data to insert
- **key** (*Callable, optional*) – sort key

Returns
sorted chunk of new data, insertion position

Return type
tuple

spinetoolbox.helpers.**load_project_dict**(*project_config_dir*, *logger*)

Loads project dictionary from project directory.

Parameters

- **project_config_dir** (*str*) – project's .spinetoolbox directory
- **logger** (LoggerInterface) – a logger

Returns
project dictionary

Return type
dict

spinetoolbox.helpers.**load_local_project_data**(*project_config_dir*, *logger*)

Loads local project data.

Parameters

- **project_config_dir** (*Path or str*) – project's .spinetoolbox directory
- **logger** (LoggerInterface) – a logger

Returns
project's local data

Return type
dict

spinetoolbox.helpers.**merge_dicts**(*source*, *target*)

Merges two dictionaries that may contain nested dictionaries recursively.

Parameters

- **source** (*dict*) – dictionary that will be merged to `target`
- **target** (*dict*) – target dictionary

spinetoolbox.helpers.**fix_lightness_color**(*color*, *lightness=240*)

spinetoolbox.helpers.**scrolling_to_bottom**(*widget*, *tolerance=1*)

spinetoolbox.helpers.**_is_metadata_item**(*item*)

Identifies a database metadata record.

Parameters
item (*dict*) – database item

Returns
True if item is metadata item, False otherwise

Return type
bool

class spinetoolbox.helpers.**HTMLTagFilter**

Bases: `html.parser.HTMLParser`

HTML tag filter.

Initialize and reset this instance.

If convert_charrefs is True (the default), all character references are automatically converted to the corresponding Unicode characters.

drain()

handle_data(*data*)

handle_starttag(*tag*, *attrs*)

spinetoolbox.helpers.**same_path**(*path1*, *path2*)

Checks if two paths are equal.

This is a lightweight version of os.path.samefile(): it doesn't check if the paths point to the same file system object but rather takes into account file system case-sensitivity and such.

Parameters

- **path1** (*str*) – a path
- **path2** (*str*) – a path

Returns
True if paths point to the same

Return type
bool

spinetoolbox.helpers.**solve_connection_file**(*connection_file*, *connection_file_dict*)

Returns the connection_file path, if it exists on this computer. If the path doesn't exist, assume that it points to a path on another computer, in which case store the contents of connection_file_dict into a tempfile.

Parameters

- **connection_file** (*str*) – Path to a connection file
- **connection_file_dict** (*dict*) –

Returns
Path to a connection file on this computer.

Return type
str

spinetoolbox.helpers.**remove_first**(*lst*, *items*)

class spinetoolbox.helpers.**SealCommand**(*command_id=1*)

Bases: PySide6.QtGui.QUndoCommand

A 'meta' command that does not store undo data but can be used in mergeWith methods of other commands.

Parameters
command_id (*int*) – command id

redo()

id()

spinetoolbox.helpers.**plain_to_rich**(*text*)

Turns plain strings into rich text.

Parameters
text (*str*) – string to convert

Returns
rich text string

Return type
str

spinetoolbox.helpers.**list_to_rich_text**(*data*)

Turns a sequence of strings into rich text list.

Parameters
data (*Sequence of str*) – iterable to convert

Returns
rich text string

Return type
str

spinetoolbox.helpers.**plain_to_tool_tip**(*text*)

Turns plain strings into rich text and empty strings/Nones to None.

Parameters
text (*str, optional*) – string to convert

Returns
rich text string or None

Return type
str or NoneType

spinetoolbox.kernel_fetcher

Contains a class for fetching kernel specs in a thread.

Module Contents

Classes

KernelFetcher	Worker class for retrieving local kernels.

class spinetoolbox.kernel_fetcher.**KernelFetcher**(*conda_path*, *fetch_mode=1*)

Bases: PySide6.QtCore.QThread

Worker class for retrieving local kernels.

Parameters

- **conda_path** (*str*) – Path to (mini)conda executable
- **fetch_mode** (*int*) – 1: Fetch all kernels, 2: Fetch regular and Conda Python kernels, 3: Fetch only regular Python kernels, 4: Fetch only regular Julia kernels, 5: Fetch kernels that are neither Python nor Julia

kernel_found

stop_fetcher

stop_thread()

Slot for handling a request to stop the thread.

get_all_regular_kernels()

Finds all kernel specs as quickly as possible.

get_all_conda_kernels()

Finds auto-generated Conda kernels.

run()

Finds kernel specs based on selected fetch mode. Sends found kernels one-by-one via signals.

static get_icon(*p*)

Retrieves the kernel's icon. First tries to find the .svg icon then .png's.

Parameters
p (*str*) – Path to Kernel's resource directory

Returns
Kernel's icon or a null icon if icon was not found.

Return type
QIcon

static get_kernel_deats(*kernel_path*)

Reads kernel.json from given kernel's resource dir and returns the details in a dictionary.

Parameters
kernel_path (*str*) – Full path to kernel resource directory

Returns
language (str), path to executable (str), display name (str), project (str) (NA for Python kernels)

Return type
dict

spinetoolbox.link

Classes for drawing graphics items on QGraphicsScene.

Module Contents

Classes

LinkBase	Base class for Link and LinkDrawer.
_IconBase	Base class for icons to show over a Link.
_SvgIcon	An SVG icon to show over a Link.
_TextIcon	A font awesome icon to show over a Link.
_WarningTextIcon	A font awesome icon to show over a Link.
JumpOrLink	Base class for Jump and Link.
Link	A graphics item to represent the connection between two project items.
JumpLink	A graphics icon to represent a jump connection between items.
LinkDrawerBase	A base class for items intended for drawing links between project items.
ConnectionLinkDrawer	An item for drawing connection links between project items.
JumpLinkDrawer	An item for drawing jump connections between project items.

Functions

_regular_polygon_points(n, side[, initial_angle])	

Attributes

LINK_COLOR

JUMP_COLOR

spinetoolbox.link.**LINK_COLOR**

spinetoolbox.link.**JUMP_COLOR**

class spinetoolbox.link.**LinkBase**(*toolbox*, *src_connector*, *dst_connector*)

Bases: PySide6.QtWidgets.QGraphicsPathItem

Base class for Link and LinkDrawer.

Mainly provides the `update_geometry` method for 'drawing' the link on the scene.

Parameters

- **toolbox** (ToolboxUI) – main UI class instance
- **src_connector** (ConnectorButton, *optional*) – Source connector button
- **dst_connector** (ConnectorButton) – Destination connector button

property outline_color

property magic_number

property src_rect

Returns the scene rectangle of the source connector.

property src_center

Returns the center point of the source rectangle.

property dst_rect

Returns the scene rectangle of the destination connector.

property dst_center

Returns the center point of the destination rectangle.

_COLOR

shape()

moveBy(*_dx*, *_dy*)

Does nothing. This item is not moved the regular way, but follows the ConnectorButtons it connects.

update_geometry(*curved_links=None*)

Updates geometry.

guide_path()

For tests.

_do_update_geometry()

Sets the path for this item.

_add_ellipse_path(*path*)

Adds an ellipse for the link's base.

Parameters
QPainterPath –

_get_joint_angle()

_add_arrow_path(*path*)

Returns an arrow path for the link's tip.

Parameters
QPainterPath –

static _get_offset(*button*)

_get_src_offset()

_get_dst_offset()

_find_new_point(*points*, *target*)

Finds a new point that approximates points to target in a smooth trajectory. Returns the new point, or None if no need for approximation.

Parameters

- **points** (*list(QPointF)*) –
- **target** (*QPointF*) –

Returns
QPointF or None

_close_enough(*p1*, *p2*)

_make_guide_path(*curved_links=False*)

Returns a 'narrow' path connecting this item's source and destination.

Parameters
curved_links (*bool*) – Whether the path should follow a curved line or just a straight line

Returns
QPainterPath

itemChange(*change*, *value*)

Wipes out the link when removed from scene.

wipe_out()

Removes any trace of this item from the system.

class spinetoolbox.link._IconBase(*x*, *y*, *w*, *h*, *parent*, *tooltip=None*, *active=True*)

Bases: PySide6.QtWidgets.QGraphicsEllipseItem

Base class for icons to show over a Link.

hoverEnterEvent(*event*)

hoverLeaveEvent(*event*)

class spinetoolbox.link._SvgIcon(*parent*, *extent*, *path*, *tooltip=None*, *active=False*)

Bases: *_IconBase*

An SVG icon to show over a Link.

wipe_out()

Cleans up icon's resources.

class spinetoolbox.link._TextIcon(*parent*, *extent*, *char*, *tooltip=None*, *active=False*)

Bases: *_IconBase*

A font awesome icon to show over a Link.

wipe_out()

Cleans up icon's resources.

class spinetoolbox.link._WarningTextIcon(*parent*, *extent*, *char*, *tooltip*)

Bases: *_TextIcon*

A font awesome icon to show over a Link.

class spinetoolbox.link.JumpOrLink(*toolbox*, *src_connector*, *dst_connector*)

Bases: *LinkBase*

Base class for Jump and Link.

Parameters

- **toolbox** (ToolboxUI) – main UI class instance
- **src_connector** (ConnectorButton, *optional*) – Source connector button
- **dst_connector** (ConnectorButton) – Destination connector button

abstract property item

_do_update_geometry()

See base class.

_place_icons()

mousePressEvent(*e*)

Ignores event if there's a connector button underneath, to allow creation of new links.

Parameters

e (*QGraphicsSceneMouseEvent*) – Mouse event

contextMenuEvent(*e*)

Selects the link and shows context menu.

Parameters

e (*QGraphicsSceneMouseEvent*) – Mouse event

paint(*painter*, *option*, *widget=None*)

Sets a dashed pen if selected.

shape()

wipe_out()

Removes any trace of this item from the system.

_make_execution_animation()

Returns an animation to play when execution 'passes' through this link.

Returns
QVariantAnimation

run_execution_animation()

Runs execution animation.

_handle_execution_animation_value_changed(*step*)

class spinetoolbox.link.**Link**(*toolbox*, *src_connector*, *dst_connector*, *connection*)

Bases: *JumpOrLink*

A graphics item to represent the connection between two project items.

Parameters

- **toolbox** (ToolboxUI) – main UI class instance
- **src_connector** (ConnectorButton) – Source connector button
- **dst_connector** (ConnectorButton) – Destination connector button
- **connection** (LoggingConnection) – connection this link represents

property name

property connection

property item

_COLOR

_MEMORY = '\uf538'

_FILTERS = '\uf0b0'

_PURGE = '\uf0e7'

_WARNING = '\uf06a'

_DATAPACKAGE = ':/icons/datapkg.svg'

update_icons()

itemChange(*change*, *value*)

Brings selected link to top.

class spinetoolbox.link.**JumpLink**(*toolbox*, *src_connector*, *dst_connector*, *jump*)

Bases: *JumpOrLink*

A graphics icon to represent a jump connection between items.

Parameters

- **toolbox** (ToolboxUI) – main UI class instance
- **src_connector** (ConnectorButton) – Source connector button
- **dst_connector** (ConnectorButton) – Destination connector button
- **jump** (*spine_engine.project_item.connection.Jump*) – connection this link represents

property jump

property item

property name

_COLOR

_ISSUE = '\uf071'

issues()

Checks if jump is well-defined.

Returns
issues regarding the jump

Return type
list of str

update_icons()

class spinetoolbox.link.LinkDrawerBase(*toolbox*)

Bases: *LinkBase*

A base class for items intended for drawing links between project items.

Parameters
toolbox (ToolboxUI) – main UI class instance

property src_rect

Returns the scene rectangle of the source connector.

property dst_rect

Returns the scene rectangle of the destination connector.

property dst_center

Returns the center point of the destination rectangle.

_get_dst_offset()

abstract add_link()

Makes link between source and destination connectors.

wake_up(*src_connector*)

Sets the source connector, shows this item and adds it to the scene. After calling this, the scene is in link drawing mode.

Parameters
src_connector (ConnectorButton) – source connector

sleep()

Removes this drawer from the scene, clears its source and destination connectors, and hides it. After calling this, the scene is no longer in link drawing mode.

class spinetoolbox.link.ConnectionLinkDrawer(*toolbox*)

Bases: *LinkDrawerBase*

An item for drawing connection links between project items.

Parameters
toolbox (ToolboxUI) – main UI class instance

_COLOR

add_link()

Makes link between source and destination connectors.

wake_up(*src_connector*)

Sets the source connector, shows this item and adds it to the scene. After calling this, the scene is in link drawing mode.

Parameters
src_connector (ConnectorButton) – source connector

sleep()

Removes this drawer from the scene, clears its source and destination connectors, and hides it. After calling this, the scene is no longer in link drawing mode.

class spinetoolbox.link.**JumpLinkDrawer**(*toolbox*)

Bases: *LinkDrawerBase*

An item for drawing jump connections between project items.

Parameters
toolbox (ToolboxUI) – main UI class instance

_COLOR

add_link()

Makes link between source and destination connectors.

spinetoolbox.link.**_regular_polygon_points**(*n*, *side*, *initial_angle=0*)

spinetoolbox.load_project_items

Functions to load project item modules.

Module Contents

Functions

load_project_items(items_package_name)	Loads project item modules.

spinetoolbox.load_project_items.**load_project_items**(*items_package_name*)

Loads project item modules.

Parameters
items_package_name (*str*) – name of the package that contains the project items

Returns

two dictionaries; first maps item type to its category
while second maps item type to item factory

Return type
tuple of dict

spinetoolbox.log_mixin

Contains LogMixin.

Module Contents

Classes

LogMixin

class spinetoolbox.log_mixin.**LogMixin**

add_log_message(*filter_id*, *message*)

Adds a message to the log document.

Parameters

- **filter_id** (*str*) – filter identifier
- **message** (*str*) – formatted message

add_event_message(*filter_id*, *msg_type*, *msg_text*)

Adds a message to the log document.

Parameters

- **filter_id** (*str*) – filter identifier
- **msg_type** (*str*) – message type
- **msg_text** (*str*) – message text

add_process_message(*filter_id*, *msg_type*, *msg_text*)

Adds a message to the log document.

Parameters

- **filter_id** (*str*) – filter identifier
- **msg_type** (*str*) – message type
- **msg_text** (*str*) – message text

spinetoolbox.logger_interface

A logger interface.

Module Contents

Classes

LoggerInterface	Placeholder for signals that can be emitted to send messages to an output device.

class spinetoolbox.logger_interface.**LoggerInterface**

Bases: PySide6.QtCore.QObject

Placeholder for signals that can be emitted to send messages to an output device.

The signals should be connected to a concrete logging system.

Currently, this is just a 'model interface'. ToolboxUI contains the same signals so it can be used as a drop-in replacement for this class.

msg

Emits a notification message.

msg_success

Emits a message on success

msg_warning

Emits a warning message.

msg_error

Emits an error message.

msg_proc

Emits a message originating from a subprocess (usually something printed to stdout).

msg_proc_error

Emits an error message originating from a subprocess (usually something printed to stderr).

information_box

Requests an 'information message box' (e.g. a message window) to be opened with a given title and message.

error_box

Requests an 'error message box' to be opened with a given title and message.

spinetoolbox.main

Provides the main() function.

Module Contents

Functions

main()	Creates main window GUI and starts main event loop.
_make_argument_parser()	Returns a command line argument parser configured for Toolbox use.
_add_pywin32_system32_to_path()	Adds a directory to PATH on Windows that is required to make pywin32 work

Attributes

dirname
plugin_path

spinetoolbox.main.**dirname**

spinetoolbox.main.**plugin_path**

spinetoolbox.main.**main**()

Creates main window GUI and starts main event loop.

spinetoolbox.main.**_make_argument_parser**()

Returns a command line argument parser configured for Toolbox use.

Returns
Toolbox' command line argument parser

Return type
ArgumentParser

spinetoolbox.main.**_add_pywin32_system32_to_path**()

Adds a directory to PATH on Windows that is required to make pywin32 work on (Conda) Python 3.8. See https://github.com/spine-tools/Spine-Toolbox/issues/1230.

spinetoolbox.metaobject

MetaObject class.

Module Contents

Classes

MetaObject	Class for an object which has a name, type, and some description.

class spinetoolbox.metaobject.**MetaObject**(*name*, *description*)

Bases: PySide6.QtCore.QObject

Class for an object which has a name, type, and some description.

Parameters

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

set_name(*name*)

Set object name and short name. Note: Check conflicts (e.g. name already exists) before calling this method.

Parameters
name (*str*) – New (long) name for this object

set_description(*description*)

Set object description.

Parameters
description (*str*) – Object description

spinetoolbox.plotting

Functions for plotting on PlotWidget.

Module Contents

Classes

PlotType	Generic enumeration.
IndexName	
XYData	Two-dimensional data for plotting.
TreeNode	A labeled node in tree structure.
ParameterTableHeaderSection	Header section info for Database editor's parameter tables.
_PlotStackedBars	

Functions

convert_indexed_value_to_tree(value)	Converts indexed values to tree nodes recursively.
turn_node_to_xy_data(root_node, y_label_position[, ...])	Constructs plottable data and indexes recursively.
raise_if_not_common_x_labels(data_list)	Raises an exception if data has different x axis labels.
raise_if_incompatible_x(data_list)	Raises an exception if the types of x data don't match.
reduce_indexes(data_list)	Removes redundant indexes from given XYData.

continues on next page

Table 2 – continued from previous page

combine_data_with_same_indexes(data_list)	Combines data with same data indexes into the same x axis.
_always_single_y_axis(plot_type)	Returns True if a single y-axis should be used.
plot_data(data_list[, plot_widget, plot_type])	Returns a plot widget with plots of the given data.
_plot_single_y_axis(data_list, y_label, axes, plot_type)	Plots all data on single y-axis.
_plot_stacked_line(data_list, y_label, axes)	Plots all data as stacked lines.
_plot_bar(data_list, y_label, axes)	Plots all data as bars.
_plot_double_y_axis(data_list, y_labels, plot_widget, ...)	Plots all data on two y-axes.
_make_x_plottable(xs)	Converts x-axis values to something matplotlib can handle.
_make_time_series_settings(plot_settings)	Creates plot settings suitable for time series step plots.
_make_plot_function(plot_type, x_data_type, axes)	Decides plot method and default keyword arguments based on XYData.
_is_time_stamp_type(data_type)	Tests if a type looks like time stamp.
_bar(x, y, axes, **kwargs)	Plots bar chart on axes but returns patches instead of bar container.
_group_bars(data_list)	Gives data with same x small offsets to prevent bar stacking.
_clear_plot(plot_widget)	Removes plots and legend from plot widget.
_limit_string_x_tick_labels(data, plot_widget)	Limits the number of x tick labels in case x-axis consists of strings.
_table_display_row(row)	Calculates a human-readable row number.
plot_parameter_table_selection(model, model_indexes, ...)	Returns a plot widget with plots of the selected indexes.
plot_value_editor_table_selection(model, model_indexes)	Returns a plot widget with plots of the selected indexes.
plot_pivot_table_selection(model, model_indexes[, ...])	Returns a plot widget with plots of the selected indexes.
plot_db_mngr_items(items, db_maps[, plot_widget])	Returns a plot widget with plots of database manager parameter value items.
_has_x_column(model, source_model)	Checks if pivot source model has x column.
_set_default_node(root_node, key, label)	Gets node from the contents of root_node adding a new node if necessary.
_get_parsed_value(model_index, display_row)	Gets parsed value from model.
_pivot_index_names(indexes)	Gathers index names from pivot table.
_pivot_display_row(row, source_model)	Calculates display row for pivot table.
_convert_to_leaf(y)	Converts parameter value to leaf TreeElement.
add_row_to_exception(row, display_row)	Adds row information to PlottingError if it is raised in the with block.
add_array_plot(plot_widget, value)	Adds an array plot to a plot widget.
add_time_series_plot(plot_widget, value)	Adds a time series step plot to a plot widget.

Attributes

LEGEND_PLACEMENT_THRESHOLD
_BASE_SETTINGS
_SCATTER_PLOT_SETTINGS
_LINE_PLOT_SETTINGS
_SCATTER_LINE_PLOT_SETTINGS

spinetoolbox.plotting.**LEGEND_PLACEMENT_THRESHOLD = 8**

class spinetoolbox.plotting.**PlotType**

Bases: enum.Enum

Generic enumeration.

Derive from this class to define new enumerations.

SCATTER

SCATTER_LINE

LINE

STACKED_LINE

BAR

STACKED_BAR

spinetoolbox.plotting.**_BASE_SETTINGS**

spinetoolbox.plotting.**_SCATTER_PLOT_SETTINGS**

spinetoolbox.plotting.**_LINE_PLOT_SETTINGS**

spinetoolbox.plotting.**_SCATTER_LINE_PLOT_SETTINGS**

exception spinetoolbox.plotting.**PlottingError**

Bases: Exception

An exception signalling failure in plotting.

Initialize self. See help(type(self)) for accurate signature.

class spinetoolbox.plotting.**IndexName**

label: str

id: int

class spinetoolbox.plotting.**XYData**

Two-dimensional data for plotting.

x: List[Union[float, int, str, numpy.datetime64]]

y: List[Union[float, int]]

x_label: *IndexName*

y_label: str

data_index: List[str]

index_names: List[*IndexName*]

class spinetoolbox.plotting.**TreeNode**

A labeled node in tree structure.

label: Union[str, *IndexName*]

content: Dict

class spinetoolbox.plotting.**ParameterTableHeaderSection**

Header section info for Database editor's parameter tables.

label: str

separator: Optional[str]

spinetoolbox.plotting.**convert_indexed_value_to_tree**(*value*)

Converts indexed values to tree nodes recursively.

Parameters
value (*IndexedValue*) – value to convert

Returns
root node of the converted tree

Return type
TreeNode

Raises
ValueError – raised when leaf value couldn't be converted to float

spinetoolbox.plotting.**turn_node_to_xy_data**(*root_node*, *y_label_position*, *index_names=None*, *indexes=None*)

Constructs plottable data and indexes recursively.

Parameters

- **root_node** (TreeNode) – root node
- **y_label_position** (*int, optional*) – position of y label in indexes
- **index_names** (*list of IndexName, optional*) – list of current index names
- **indexes** (*list*) – list of current indexes

Yields
XYData – plot data

spinetoolbox.plotting.**raise_if_not_common_x_labels**(*data_list*)

Raises an exception if data has different x axis labels.

Parameters
data_list (*list of XYData*) – data to check

Raises
PlottingError – raised if x axis labels don't match.

spinetoolbox.plotting.**raise_if_incompatible_x**(*data_list*)

Raises an exception if the types of x data don't match.

Parameters
data_list (*list of XYData*) – data to check

Raises
PlottingError – raised if x data types don't match.

spinetoolbox.plotting.**reduce_indexes**(*data_list*)

Removes redundant indexes from given XYData.

Parameters
data_list (*list of XYData*) – data to reduce

Returns
reduced data list and list of common data indexes

Return type
tuple

spinetoolbox.plotting.**combine_data_with_same_indexes**(*data_list*)

Combines data with same data indexes into the same x axis.

Parameters
data_list (*list of XYData*) – data to combine

Returns
combined data

Return type
list of XYData

spinetoolbox.plotting.**_always_single_y_axis**(*plot_type*)

Returns True if a single y-axis should be used.

Parameters
plot_type (PlotType) – plot type

Returns
True if single y-axis is required, False otherwise

Return type
bool

spinetoolbox.plotting.**plot_data**(*data_list*, *plot_widget=None*, *plot_type=None*)

Returns a plot widget with plots of the given data.

Parameters

- **data_list** (*list of XYData*) – data to plot
- **plot_widget** (PlotWidget, *optional*) – an existing plot widget to draw into or None to create a new widget
- **plot_type** (PlotType, *optional*) – plot type

Returns
a PlotWidget object

spinetoolbox.plotting.**_plot_single_y_axis**(*data_list*, *y_label*, *axes*, *plot_type*)

Plots all data on single y-axis.

Parameters

- **data_list** (*list of XYData*) – data to plot
- **y_label** (*str*) – y-axis label
- **axes** (*Axes*) – plot axes
- **plot_type** (PlotType) – plot type

Returns
legend handles

Return type
list

spinetoolbox.plotting.**_plot_stacked_line**(*data_list*, *y_label*, *axes*)

Plots all data as stacked lines.

Parameters

- **data_list** (*list of XYData*) – data to plot
- **y_label** (*str*) – y-axis label
- **axes** (*Axes*) – plot axes

Returns
legend handles

Return type
list

spinetoolbox.plotting.**_plot_bar**(*data_list*, *y_label*, *axes*)

Plots all data as bars.

Parameters

- **data_list** (*list of XYData*) – data to plot
- **y_label** (*str*) – y-axis label
- **axes** (*Axes*) – plot axes

Returns
legend handles

Return type
list

spinetoolbox.plotting.**_plot_double_y_axis**(*data_list*, *y_labels*, *plot_widget*, *plot_type*)

Plots all data on two y-axes.

Parameters

- **data_list** (*list of XYData*) – data to plot
- **y_labels** (*list of str*) – y-axis labels
- **plot_widget** (PlotWidget) – plot widget
- **plot_type** (PlotType) – plot type

Returns
legend handles

Return type
list

spinetoolbox.plotting.**_make_x_plottable**(*xs*)

Converts x-axis values to something matplotlib can handle.

Parameters
xs (*list*) – x values

Returns
x values

Return type
list

class spinetoolbox.plotting.**_PlotStackedBars**(*axes*)

__call__(*x*, *height*, ***kwargs*)

spinetoolbox.plotting.**_make_time_series_settings**(*plot_settings*)

Creates plot settings suitable for time series step plots.

Parameters
plot_settings (*dict*) – base plot settings

Returns
time series step plot settings

Return type
dict

spinetoolbox.plotting.**_make_plot_function**(*plot_type*, *x_data_type*, *axes*)

Decides plot method and default keyword arguments based on XYData.

Parameters

- **plot_type** (PlotType) – plot type
- **x_data_type** (*Type*) – data type of x-axis
- **axes** (*Axes*) – plot axes

Returns
plot method

Return type
Callable

spinetoolbox.plotting.**_is_time_stamp_type**(*data_type*)

Tests if a type looks like time stamp.

Parameters
data_type (*Type*) – data type to test

Returns
True if type is a time stamp type, False otherwise

Return type
bool

spinetoolbox.plotting.**_bar**(*x*, *y*, *axes*, ***kwargs*)

Plots bar chart on axes but returns patches instead of bar container.

Parameters

- **x** (*Any*) – x data
- **y** (*Any*) – y data
- **axes** (*Axes*) – plot axes
- ****kwargs** – keyword arguments passed to bar()

Returns
patches

Return type
list of Patch

spinetoolbox.plotting.**_group_bars**(*data_list*)

Gives data with same x small offsets to prevent bar stacking.

Parameters
data_list (*List of XYData*) – squeezed data

Returns
grouped data, bar width and x ticks

Return type
tuple

spinetoolbox.plotting.**_clear_plot**(*plot_widget*)

Removes plots and legend from plot widget.

Parameters
plot_widget (PlotWidget) – plot widget

spinetoolbox.plotting.**_limit_string_x_tick_labels**(*data*, *plot_widget*)

Limits the number of x tick labels in case x-axis consists of strings.

Matplotlib tries to plot every single x tick label if they are strings. This can become very slow if the labels are numerous.

Parameters

- **data** (*list of XYData*) – plot data
- **plot_widget** (PlotWidget) – plot widget

spinetoolbox.plotting.**_table_display_row**(*row*)

Calculates a human-readable row number.

Parameters
row (*int*) – model row

Returns
row number

Return type
int

spinetoolbox.plotting.**plot_parameter_table_selection**(*model*, *model_indexes*, *table_header_sections*, *value_section_label*, *plot_widget=None*)

Returns a plot widget with plots of the selected indexes.

Parameters

- **model** (*QAbstractTableModel*) – a model
- **model_indexes** (*Iterable of QModelIndex*) – a list of QModelIndex objects for plotting
- **table_header_sections** (*list of ParameterTableHeaderSection*) – table header labels
- **value_section_label** (*str*) – value column's header label
- **plot_widget** (PlotWidget, *optional*) – an existing plot widget to draw into or None to create a new widget

Returns
a PlotWidget object

Return type
PlotWidget

spinetoolbox.plotting.**plot_value_editor_table_selection**(*model*, *model_indexes*, *plot_widget=None*)

Returns a plot widget with plots of the selected indexes.

Parameters

- **model** (*QAbstractTableModel*) – a model
- **model_indexes** (*Iterable of QModelIndex*) – a list of QModelIndex objects for plotting
- **plot_widget** (PlotWidget, *optional*) – an existing plot widget to draw into or None to create a new widget

Returns
a PlotWidget object

Return type
PlotWidget

spinetoolbox.plotting.**plot_pivot_table_selection**(*model*, *model_indexes*, *plot_widget=None*)

Returns a plot widget with plots of the selected indexes.

Parameters

- **model** (*QAbstractTableModel*) – a model
- **model_indexes** (*Iterable of QModelIndex*) – a list of QModelIndex objects for plotting
- **plot_widget** (PlotWidget, *optional*) – an existing plot widget to draw into or None to create a new widget

Returns
a PlotWidget object

Return type
PlotWidget

spinetoolbox.plotting.**plot_db_mngr_items**(*items*, *db_maps*, *plot_widget=None*)

Returns a plot widget with plots of database manager parameter value items.

Parameters

- **items** (*list of dict*) – parameter value items

- **db_maps** (*list of DatabaseMapping*) – database mappings corresponding to items
- **plot_widget** (PlotWidget, *optional*) – widget to add plots to

spinetoolbox.plotting.**_has_x_column**(*model*, *source_model*)

Checks if pivot source model has x column.

Parameters

- **model** (PivotTableSortFilterProxy) – proxy pivot model
- **source_model** (PivotTableModelBase) – pivot table model

Returns
True if x pivot table has column, False otherwise

Return type
bool

spinetoolbox.plotting.**_set_default_node**(*root_node*, *key*, *label*)

Gets node from the contents of root_node adding a new node if necessary.

Parameters

- **root_node** (TreeNode) – root node
- **key** (*Hashable*) – key to root_node contents
- **label** (*str*) – label of possible new node

Returns
node at given key

Return type
TreeNode

spinetoolbox.plotting.**_get_parsed_value**(*model_index*, *display_row*)

Gets parsed value from model.

Parameters

- **model_index** (*QModelIndex*) – model index
- **display_row** (*Callable*) – callable that returns a display row

Returns
parsed value

Return type
Any

Raises
PlottingError – raised if parsing of value failed

spinetoolbox.plotting.**_pivot_index_names**(*indexes*)

Gathers index names from pivot table.

Parameters
indexes (*tuple of str*) – "path" of indexes

Returns
names corresponding to given indexes

Return type
tuple of str

spinetoolbox.plotting.**_pivot_display_row**(*row*, *source_model*)

Calculates display row for pivot table.

Parameters

- **row** (*int*) – row in source table model
- **source_model** (*QAbstractItemModel*) – pivot model

Returns
human-readable row number

Return type
int

spinetoolbox.plotting.**_convert_to_leaf**(*y*)

Converts parameter value to leaf TreeElement.

Parameters
y (*Any*) – parameter value

Returns
leaf element

Return type
float or datetime or *TreeNode*

spinetoolbox.plotting.**add_row_to_exception**(*row*, *display_row*)

Adds row information to PlottingError if it is raised in the with block.

Parameters

- **row** (*int*) – row
- **display_row** (*Callable*) – function to convert row to display row

spinetoolbox.plotting.**add_array_plot**(*plot_widget*, *value*)

Adds an array plot to a plot widget.

Parameters

- **plot_widget** (PlotWidget) – a plot widget to modify
- **value** (*Array*) – the array to plot

spinetoolbox.plotting.**add_time_series_plot**(*plot_widget*, *value*)

Adds a time series step plot to a plot widget.

Parameters

- **plot_widget** (PlotWidget) – a plot widget to modify
- **value** (*TimeSeries*) – the time series to plot

spinetoolbox.plugin_manager

Contains PluginManager class.

Module Contents

Classes

PluginManager	Class for managing plugins.
_PluginWorker	

Functions

_download_file(remote, local)	
_download_plugin(plugin, plugin_local_dir)	

spinetoolbox.plugin_manager.**_download_file**(*remote*, *local*)

spinetoolbox.plugin_manager.**_download_plugin**(*plugin*, *plugin_local_dir*)

class spinetoolbox.plugin_manager.**PluginManager**(*toolbox*)

Class for managing plugins.

Parameters
toolbox (ToolboxUI) – Toolbox instance.

property plugin_toolbars

property plugin_specs

load_installed_plugins()

Loads installed plugins and adds their specifications to toolbars.

reload_plugins_with_local_data()

Reloads plugins that have project specific local data.

load_individual_plugin(*plugin_dir*, *specification_local_data*)

Loads plugin from directory.

Parameters

- **plugin_dir** (*str*) – path of plugin dir with "plugin.json" in it.
- **specification_local_data** (*dict*) – specification local data

_create_worker()

_clean_up_worker(*worker*)

_load_registry()

show_install_plugin_dialog(*_=False*)

_do_show_install_plugin_dialog()

_install_plugin(*plugin_name*)

Installs plugin from the registry and loads it.

Parameters
plugin_name (*str*) – plugin name

_load_installed_plugin(*plugin_local_dir*)

show_manage_plugins_dialog(*_=False*)

_do_show_manage_plugins_dialog()

_remove_plugin(*plugin_name*)

Removes installed plugin.

Parameters
plugin_name (*str*) – plugin name

_update_plugin(*plugin_name*)

exception spinetoolbox.plugin_manager.**PluginWorkFailed**

Bases: Exception

Exception to signal plugin worker that something failed.

Initialize self. See help(type(self)) for accurate signature.

class spinetoolbox.plugin_manager.**_PluginWorker**

Bases: PySide6.QtCore.QObject

failed

finished

succeeded

start(*function*, **args*, ***kwargs*)

_do_work()

clean_up()

spinetoolbox.project

Spine Toolbox project class.

Module Contents

Classes

ItemNameStatus	Generic enumeration.
SpineToolboxProject	Class for Spine Toolbox projects.

Functions

node_successors(g)	Returns a dict mapping nodes in topological order to a list of successors.
_ranks(node_successors)	Calculates node ranks.

class spinetoolbox.project.**ItemNameStatus**

Bases: enum.Enum

Generic enumeration.

Derive from this class to define new enumerations.

OK

INVALID

EXISTS

SHORT_NAME_EXISTS

class spinetoolbox.project.**SpineToolboxProject**(*toolbox*, *p_dir*, *plugin_specs*, *app_settings*, *settings*, *logger*)

Bases: *spinetoolbox.metaobject.MetaObject*

Class for Spine Toolbox projects.

Parameters

- **toolbox** (ToolboxUI) – toolbox of this project
- **p_dir** (*str*) – Project directory
- **plugin_specs** (*Iterable of ProjectItemSpecification*) – specifications available as plugins
- **app_settings** (*QSettings*) – Toolbox settings
- **settings** (ProjectSettings) – project settings
- **logger** (LoggerInterface) – a logger instance

property all_item_names

property n_items

property settings

property connections

property app_settings

project_about_to_be_torn_down
: Emitted before project is being torn down.

project_execution_about_to_start
: Emitted just before the entire project is executed.

project_execution_finished
: Emitted after the entire project execution finishes.

connection_established
: Emitted after new connection has been added to project.

connection_about_to_be_removed
: Emitted before connection removal.

connection_updated
: Emitted after a connection has been updated.

jump_added
: Emitted after a jump has been added.

jump_about_to_be_removed
: Emitted before a jump is removed.

jump_updated
: Emitted after a jump has been replaced by another.

item_added
: Emitted after a project item has been added.

item_about_to_be_removed
: Emitted before project item removal.

item_renamed
: Emitted after project item has been renamed.

specification_added
: Emitted after a specification has been added.

specification_about_to_be_removed
: Emitted before a specification will be removed.

specification_replaced
: Emitted after a specification has been replaced.

specification_saved
: Emitted after a specification has been saved.

LOCAL_EXECUTION_JOB_ID = '1'

toolbox()
: Returns Toolbox main window.

  Returns
  main window

  Return type
  ToolboxUI

has_items()

Returns True if project has project items.

Returns
True if project has items, False otherwise

Return type
bool

get_item(*name*)

Returns project item.

Parameters
name (*str*) – Item's name

Returns
Project item

Return type
ProjectItem

get_items()

Returns all project items.

Returns
All project items

Return type
list of ProjectItem

get_items_by_type(*_type*)

Returns all project items with given _type.

Parameters
_type (*str*) – Project Item type

Returns
Project Items with given type or an empty list if none found

Return type
list of ProjectItem

_create_project_structure(*directory*)

Makes the given directory a Spine Toolbox project directory. Creates directories and files that are common to all projects.

Parameters
directory (*str*) – Abs. path to a directory that should be made into a project directory

Returns
True if project structure was created successfully, False otherwise

Return type
bool

call_set_description(*description*)

set_description(*description*)

Set object description.

Parameters
description (*str*) – Object description

save()

Collects project information and objects into a dictionary and writes it to a JSON file.

_save_all_specifications(*local_path*)

Writes all specifications except plugins to disk.

Local specification data is also written to disk, including local data from plugins.

Parameters
local_path (*Path*) –

Returns
specification local data that is supposed to be stored in a project specific place

Return type
dict

_pop_local_data_from_items_dict(*items_dict*)

Pops local data from project items dict.

Parameters
items_dict (*dict*) – items dict

Returns
local project item data

Return type
dict

static _dump(*target_dict*, *out_stream*)

Dumps given dict into output stream.

Parameters

- **target_dict** (*dict*) – dictionary to dump
- **out_stream** (*IOBase*) – output stream

load(*spec_factories*, *item_factories*)

Loads project from its project directory.

Parameters

- **spec_factories** (*dict*) – Dictionary mapping specification name to ProjectItemSpecificationFactory
- **item_factories** (*dict*) – mapping from item type to ProjectItemFactory

Returns
True if the operation was successful, False otherwise

Return type
bool

static _merge_local_data_to_project_info(*local_data_dict*, *project_info*)

Merges local data into project info.

Parameters

- **local_data_dict** (*dict*) – local data
- **project_info** (*dict*) – project dict

connection_from_dict(*connection_dict*)

jump_from_dict(*jump_dict*)

add_specification(*specification*, *save_to_disk=True*)

Adds a specification to the project.

Parameters

- **specification** (*ProjectItemSpecification*) – specification to add
- **save_to_disk** (*bool*) – if True, save the specification to disk

Returns
A unique identifier for the specification or None if the operation was unsuccessful

Return type
int

is_specification_name_reserved(*name*)

Checks if specification exists.

Parameters
name (*str*) – specification's name

Returns
True if project has given specification, False otherwise

Return type
bool

specifications()

Yields project's specifications.

Yields
ProjectItemSpecification – specification

_specification_id()

Creates an id for specification.

Returns
new id

Return type
int

get_specification(*name_or_id*)

Returns project item specification.

Parameters
name_or_id (*str or int*) – specification's name or id

Returns
specification or None if specification was not found

Return type
ProjectItemSpecification

specification_name_to_id(*name*)

Returns identifier for named specification.

Parameters
name (*str*) – specification's name

Returns
specification's id or None if no such specification exists

Return type
int

`remove_specification`(*id_or_name*)

Removes a specification from project.

Parameters
id_or_name (*int or str*) – specification's id or name

`replace_specification`(*name*, *specification*, *save_to_disk=True*)

Replaces an existing specification.

Refreshes the spec in all items that use it.

Parameters

- **name** (*str*) – name of the specification to replace
- **specification** (*ProjectItemSpecification*) – a specification
- **save_to_disk** (*bool*) – If True, saves the given specification to disk

Returns
True if operation was successful, False otherwise

Return type
bool

`save_specification_file`(*specification*, *previous_name=None*)

Saves the given project item specification.

Save path is determined by specification directory and specification's name.

Parameters

- **specification** (*ProjectItemSpecification*) – specification to save
- **previous_name** (*str, optional*) – specification's earlier name if it has been renamed/replaced

Returns
True if operation was successful, False otherwise

Return type
bool

`_update_specification_local_data_store`(*specification*, *local_data*, *previous_name*)

Updates the file containing local data of project's specifications.

Parameters

- **specification** (*ProjectItemSpecification*) – specification
- **local_data** (*dict*) – local data serialized into dict
- **previous_name** (*str, optional*) – specification's earlier name if it has been renamed/replaced

`_default_specification_file_path`(*specification*)

Determines a path inside project directory to save a specification.

Parameters
specification (*ProjectItemSpecification*) – specification

Returns
valid path or None if operation failed

Return type
str

add_item(*item*)

Adds a project item to project.

Parameters
item (ProjectItem) – item to add

rename_item(*previous_name*, *new_name*, *rename_data_dir_message*)

Renames a project item

Parameters

- **previous_name** (*str*) – item's current name
- **new_name** (*str*) – item's new name
- **rename_data_dir_message** (*str*) – message to show when renaming item's data directory

Returns
True if item was renamed successfully, False otherwise

Return type
bool

validate_project_item_name(*name*)

Validates item name.

Parameters
name (*str*) – proposed project item's name

Returns
validation result

Return type
ItemNameStatus

find_connection(*source_name*, *destination_name*)

Searches for a connection between given items.

Parameters

- **source_name** (*str*) – source item's name
- **destination_name** (*str*) – destination item's name

Returns
connection instance or None if there is no connection

Return type
Connection

connections_for_item(*item_name*)

Returns connections that have given item as source or destination.

Parameters
item_name (*str*) – item's name

Returns
connections connected to item

Return type
list of Connection

add_connection(**args*, *silent=False*, *notify_resource_changes=True*)

Adds a connection to the project.

A single argument is expected to be the `Logging connection` instance. Optionally, source name and position, and destination name and position can be provided instead.

Parameters

- ***args** – connection to add
- **silent** (*bool*) – If False, prints 'Link establ…' msg to Event Log
- **notify_resource_changes** (*bool*) – If True, updates resources of successor and predecessor items

Returns
True if connection was added successfully, False otherwise

Return type
bool

_notify_rsrc_changes(*destination*, *source*)

Notifies connection destination and connection source item that resources may have changed.

Parameters

- **destination** (ProjectItem) – Destination item
- **source** (ProjectItem) – Source item

remove_connection(*connection*)

Removes a connection from the project.

Parameters
connection (LoggingConnection) – connection to remove

update_connection(*connection*, *source_position*, *destination_position*)

Updates existing connection between items.

Updating does not trigger any updates to the DAG or project items.

Parameters

- **connection** (LoggingConnection) – connection to update
- **source_position** (*str*) – link's position on source item's icon
- **destination_position** (*str*) – link's position on destination item's icon

jumps_for_item(*item_name*)

Returns jumps that have given item as source or destination.

Parameters
item_name (*str*) – item's name

Returns
jumps connected to item

Return type
list of Jump

add_jump(*jump*, *silent=False*)

Adds a jump to project.

Parameters

- **jump** (*Jump*) – jump to add
- **silent** (*bool*) – if True, don't log messages

find_jump(*source_name*, *destination_name*)

Searches for a jump between given items.

Parameters

- **source_name** (*str*) – source item's name
- **destination_name** (*str*) – destination item's name

Returns
connection instance or None if there is no jump

Return type
Jump

remove_jump(*jump*)

Removes a jump from the project.

Parameters
jump (*Jump*) – jump to remove

update_jump(*jump*, *source_position*, *destination_position*)

Updates an existing jump between items.

Parameters

- **jump** (LoggingJump) – jump to update
- **source_position** (*str*) – link's position on source item's icon
- **destination_position** (*str*) – link's position on destination item's icon

_update_jump_icons()

Updates icons for all jumps in the project.

jump_issues(*jump*)

Checks if jump is OK.

Parameters
jump (*Jump*) – jump to check

Returns
list of issues, if any

Return type
list of str

_dag_iterator()

Iterates directed graphs in the project.

Yields
DiGraph

dag_with_node(*node*)

Returns the DiGraph that contains the given node (project item) name (str).

restore_project_items(*items_dict*, *item_factories*)

Restores project items from dictionary.

Parameters

- **items_dict** (*dict*) – a mapping from item name to item dict
- **item_factories** (*dict*) – a mapping from item type to ProjectItemFactory

remove_item_by_name(*item_name*, *delete_data=False*)

Removes project item by its name.

Parameters

- **item_name** (*str*) – Item's name
- **delete_data** (*bool*) – If set to True, deletes the directories and data associated with the item

execute_dags(*dags*, *execution_permits_list*, *msg*)

Executes given dags.

Parameters

- **dags** (*list of DiGraph*) – List of DAGs
- **execution_permits_list** (*list of dict*) –
- **msg** (*str*) – Message to log before execution

_execute_dags(*dags*, *execution_permits_list*)

create_engine_worker(*dag*, *execution_permits*, *dag_identifier*, *settings*, *job_id*)

Creates and returns a SpineEngineWorker to execute given *validated* dag.

Parameters

- **dag** (*DiGraph*) – The dag
- **execution_permits** (*dict*) – mapping item names to a boolean indicating whether to execute it or skip it
- **dag_identifier** (*str*) – A string identifying the dag, for logging
- **settings** (*dict*) – project and app settings to send to the spine engine.
- **job_id** (*str*) – job id

Returns
SpineEngineWorker

_handle_engine_worker_finished(*worker*)

execute_selected(*names*)

Executes DAGs corresponding to given project items.

Parameters
names (*Iterable of str*) – Names of selected items

_split_to_subdags(*dag*, *selected_items*)

Checks if given dag contains weakly connected components. If it does, the weakly connected components are split into their own (sub)dags. If not, the dag is returned unaltered.

Parameters

- **dag** (*DiGraph*) – Dag that is checked if it needs to be split into subdags
- **selected_items** (*list*) – Names of selected items

Returns
List of dags, ready for execution

Return type
list of DiGraph

execute_project()

Executes all dags in the project.

_validate_dags(*dags*)

Validates dags and logs error messages.

Parameters
dags (*Iterable*) – dags to validate

Returns
validated dag

Return type
list

stop()

Stops execution.

notify_resource_changes_to_predecessors(*item*)

Updates resources for direct predecessors of given item.

Parameters
item (ProjectItem) – item whose resources have changed

_update_incoming_connection_and_jump_resources(*item_name*, *trigger_resources*)

notify_resource_changes_to_successors(*item*)

Updates resources for direct successors and outgoing connections of given item.

Parameters
item (ProjectItem) – item whose resources have changed

_update_outgoing_connection_and_jump_resources(*item_name*, *trigger_resources*)

_notify_resource_changes(*trigger_name*, *target_names*, *provider_connections*, *update_resources*, *trigger_resources*)

Updates resources in given direction for immediate neighbours of an item.

Parameters

- **trigger_name** (*str*) – item whose resources have changed

- **target_names** (*Iterable of str*) – items to be notified
- **provider_connections** (*Callable*) – function that receives a target item name and returns a list of Connections from resource providers
- **update_resources** (*Callable*) – function that takes an item name, a list of provider names, and a dictionary of resources, and does the updating
- **trigger_resources** (*list of ProjectItemResource*) – resources from the trigger item

notify_resource_replacement_to_successors(*item*, *old*, *new*)

Replaces resources for direct successors and outgoing connections of given item.

Parameters

- **item** (ProjectItem) – item whose resources have changed
- **old** (*list of ProjectItemResource*) – old resource
- **new** (*list of ProjectItemResource*) – new resource

notify_resource_replacement_to_predecessors(*item*, *old*, *new*)

Replaces resources for direct predecessors.

Parameters

- **item** (ProjectItem) – item whose resources have changed
- **old** (*list of ProjectItemResource*) – old resources
- **new** (*list of ProjectItemResource*) – new resources

_update_item_resources(*target_item*, *direction*)

Updates up or downstream resources for a single project item. Called in both directions after removing a Connection.

Parameters

- **target_item** (ProjectItem) – item whose resource need update
- **direction** (*ExecutionDirection*) – FORWARD updates resources from upstream, BACKWARD from downstream

predecessor_names(*name*)

Collects direct predecessor item names.

Parameters
name (*str*) – name of the project item whose predecessors to collect

Returns
direct predecessor names

Return type
set of str

successor_names(*name*)

Collects direct successor item names.

Parameters
name (*str*) – name of the project item whose successors to collect

Returns
direct successor names

Return type
set of str

`descendant_names`(*name*)

Yields descendant item names.

Parameters
name (*str*) – name of the project item whose descendants to collect

Yields
str – descendant name

`outgoing_connections`(*name*)

Collects outgoing connections.

Parameters
name (*str*) – name of the project item whose connections to collect

Returns
outgoing connections

Return type
set of Connection

`_outgoing_jumps`(*name*)

Collects outgoing jumps.

Parameters
name (*str*) – name of the project item whose jumps to collect

Returns
outgoing jumps

Return type
set of Jump

`_outgoing_connections_and_jumps`(*name*)

Collects outgoing connections and jumps.

Parameters
name (*str*) – name of the project item whose connections and jumps to collect

Returns
outgoing connections and jumps

Return type
set of Connection/Jump

`incoming_connections`(*name*)

Collects incoming connections.

Parameters
name (*str*) – name of the project item whose connections to collect

Returns
incoming connections

Return type
set of Connection

`_incoming_jumps`(*name*)

Collects incoming jumps.

Parameters
name (*str*) – name of the project item whose jumps to collect

Returns
incoming jumps

Return type
set of Jump

_incoming_connections_and_jumps(*name*)

Collects incoming connections and jumps.

Parameters
name (*str*) – name of the project item whose connections and jumps to collect

Returns
incoming connections

Return type
set of Connection/Jump

_update_successor(*successor*, *incoming_connections*, *resource_cache*)

_update_predecessor(*predecessor*, *outgoing_connections*, *resource_cache*)

_is_dag_valid(*dag*)

_update_ranks(*dag*)

prepare_remote_execution()

Pings the server and sends the project as a zip-file to server.

Returns

Job Id if server is ready for remote execution, empty string if something went wrong
or LOCAL_EXECUTION_JOB_ID if local execution is enabled.

Return type
str

finalize_remote_execution(*job_id*)

Sends a request to server to remove the project directory. In addition, removes the project ZIP file from client machine.

Parameters
job_id (*str*) – job id

tear_down()

Cleans up project.

spinetoolbox.project.**node_successors**(*g*)

Returns a dict mapping nodes in topological order to a list of successors.

Parameters
g (*DiGraph*) –

Returns
dict

spinetoolbox.project.**_ranks**(*node_successors*)

Calculates node ranks.

Parameters
node_successors (*dict*) – a mapping from successor name to a list of predecessor names

Returns
a mapping from node name to rank

Return type
dict

spinetoolbox.project_commands

QUndoCommand subclasses for modifying the project.

Module Contents

Classes

SpineToolboxCommand	
SetItemSpecificationCommand	Command to set the specification for a project item.
MoveIconCommand	Command to move icons in the Design view.
SetProjectDescriptionCommand	Command to set the project description.
AddProjectItemsCommand	Command to add items.
RemoveAllProjectItemsCommand	Command to remove all items from project.
RemoveProjectItemsCommand	Command to remove items.
RenameProjectItemCommand	Command to rename project items.
AddConnectionCommand	Command to add connection between project items.
RemoveConnectionsCommand	Command to remove links.
AddJumpCommand	Command to add a jump between project items.
RemoveJumpsCommand	Command to remove jumps.
SetJumpConditionCommand	Command to set jump condition.
UpdateJumpCmdLineArgsCommand	Command to update Jump command line args.
SetFiltersOnlineCommand	Command to toggle filter value.
SetConnectionDefaultFilterOnlineStatus	Command to set connection's default filter online status.
SetConnectionFilterTypeEnabled	Command to enable and disable connection's filter types.
SetConnectionOptionsCommand	Command to set connection options.
AddSpecificationCommand	Command to add item specification to a project.
ReplaceSpecificationCommand	Command to replace item specification in project.
RemoveSpecificationCommand	Command to remove specs from a project.
SaveSpecificationAsCommand	Command to remove item specs from a project.

class spinetoolbox.project_commands.**SpineToolboxCommand**

Bases: PySide6.QtGui.QUndoCommand

property is_critical

Returns True if this command needs to be undone before closing the project without saving changes.

successfully_undone = False

Flag to register the outcome of undoing a critical command, so toolbox can react afterwards.

class spinetoolbox.project_commands.**SetItemSpecificationCommand**(*item_name*, *spec*, *old_spec*, *project*)

Bases: *SpineToolboxCommand*

Command to set the specification for a project item.

Parameters

- **item_name** (*str*) – item's name
- **spec** (*ProjectItemSpecification*) – the new spec
- **old_spec** (*ProjectItemSpecification*) – the old spec
- **project** (SpineToolboxProject) – project

redo()

undo()

class spinetoolbox.project_commands.**MoveIconCommand**(*icon*, *project*)

Bases: *SpineToolboxCommand*

Command to move icons in the Design view.

Parameters

- **icon** (ProjectItemIcon) – the icon
- **project** (SpineToolboxProject) – project

redo()

undo()

_move_to(*positions*)

class spinetoolbox.project_commands.**SetProjectDescriptionCommand**(*project*, *description*)

Bases: *SpineToolboxCommand*

Command to set the project description.

Parameters

- **project** (SpineToolboxProject) – the project
- **description** (*str*) – The new description

redo()

undo()

class spinetoolbox.project_commands.**AddProjectItemsCommand**(*project*, *items_dict*, *item_factories*)

Bases: *SpineToolboxCommand*

Command to add items.

Parameters

- **project** (SpineToolboxProject) – the project
- **items_dict** (*dict*) – a mapping from item name to item dict
- **item_factories** (*dict*) – a mapping from item type to ProjectItemFactory
- **silent** (*bool*) – If True, suppress messages

redo()

undo()

class spinetoolbox.project_commands.**RemoveAllProjectItemsCommand**(*project*, *item_factories*, *delete_data=False*)

Bases: *SpineToolboxCommand*

Command to remove all items from project.

Parameters

- **project** (SpineToolboxProject) – the project
- **item_factories** (*dict*) – a mapping from item type to ProjectItemFactory
- **delete_data** (*bool*) – If True, deletes the directories and data associated with the items

redo()

undo()

class spinetoolbox.project_commands.**RemoveProjectItemsCommand**(*project*, *item_factories*, *item_names*, *delete_data=False*)

Bases: *SpineToolboxCommand*

Command to remove items.

Parameters

- **project** (SpineToolboxProject) – The project
- **item_factories** (*dict*) – a mapping from item type to ProjectItemFactory
- **item_names** (*list of str*) – Item names
- **delete_data** (*bool*) – If True, deletes the directories and data associated with the item

redo()

undo()

class spinetoolbox.project_commands.**RenameProjectItemCommand**(*project*, *previous_name*, *new_name*)

Bases: *SpineToolboxCommand*

Command to rename project items.

Parameters

- **project** (SpineToolboxProject) – the project
- **previous_name** (*str*) – item's previous name
- **new_name** (*str*) – the new name

property is_critical

Returns True if this command needs to be undone before closing the project without saving changes.

redo()

undo()

class spinetoolbox.project_commands.**AddConnectionCommand**(*project*, *source_name*, *source_position*, *destination_name*, *destination_position*)

Bases: *SpineToolboxCommand*

Command to add connection between project items.

Parameters

- **project** (SpineToolboxProject) – project
- **source_name** (*str*) – source item's name
- **source_position** (*str*) – link's position on source item's icon
- **destination_name** (*str*) – destination item's name
- **destination_position** (*str*) – link's position on destination item's icon

redo()

undo()

class spinetoolbox.project_commands.**RemoveConnectionsCommand**(*project*, *connections*)

Bases: *SpineToolboxCommand*

Command to remove links.

Parameters

- **project** (SpineToolboxProject) – project
- **connections** (*list of LoggingConnection*) – the connections

redo()

undo()

class spinetoolbox.project_commands.**AddJumpCommand**(*project*, *source_name*, *source_position*, *destination_name*, *destination_position*)

Bases: *SpineToolboxCommand*

Command to add a jump between project items.

Parameters

- **project** (SpineToolboxProject) – project
- **source_name** (*str*) – source item's name
- **source_position** (*str*) – link's position on source item's icon
- **destination_name** (*str*) – destination item's name
- **destination_position** (*str*) – link's position on destination item's icon

redo()

undo()

class spinetoolbox.project_commands.**RemoveJumpsCommand**(*project*, *jumps*)

Bases: *SpineToolboxCommand*

Command to remove jumps.

Parameters

- **project** (SpineToolboxProject) – project
- **jumps** (*list of LoggingJump*) – the jumps

redo()

undo()

class spinetoolbox.project_commands.**SetJumpConditionCommand**(*project*, *jump*, *jump_properties*, *condition*)

Bases: *SpineToolboxCommand*

Command to set jump condition.

Parameters

- **project** (SpineToolboxProject) – project
- **jump** (*Jump*) – target jump
- **jump_properties** (JumpPropertiesWidget) – jump's properties tab
- **condition** (*dict*) – jump condition

redo()

undo()

class spinetoolbox.project_commands.**UpdateJumpCmdLineArgsCommand**(*project*, *jump*, *jump_properties*, *cmd_line_args*)

Bases: *SpineToolboxCommand*

Command to update Jump command line args.

Parameters

- **project** (SpineToolboxProject) – project
- **jump** (*Jump*) – jump
- **jump_properties** (JumpPropertiesWidget) – the item
- **cmd_line_args** (*list*) – list of command line args

redo()

undo()

class spinetoolbox.project_commands.**SetFiltersOnlineCommand**(*project*, *connection*, *resource*, *filter_type*, *online*)

Bases: *SpineToolboxCommand*

Command to toggle filter value.

Parameters

- **project** (SpineToolboxProject) – project
- **connection** (*Connection*) – connection
- **resource** (*str*) – resource label
- **filter_type** (*str*) – filter type identifier
- **online** (*dict*) – mapping from scenario/tool id to online flag

redo()

undo()

class spinetoolbox.project_commands.**SetConnectionDefaultFilterOnlineStatus**(*project*, *connection*, *default_status*)

Bases: *SpineToolboxCommand*

Command to set connection's default filter online status.

Parameters

- **project** (SpineToolboxProject) – project
- **connection** (LoggingConnection) – connection
- **default_status** (*bool*) – default filter online status

redo()

undo()

class spinetoolbox.project_commands.**SetConnectionFilterTypeEnabled**(*project*, *connection*, *filter_type*, *enabled*)

Bases: *SpineToolboxCommand*

Command to enable and disable connection's filter types.

Parameters

- **project** (SpineToolboxProject) – project
- **connection** (LoggingConnection) – connection
- **filter_type** (*str*) – filter type
- **enabled** (*bool*) – whether filter type is enabled

redo()

undo()

class spinetoolbox.project_commands.**SetConnectionOptionsCommand**(*project*, *connection*, *options*)

Bases: *SpineToolboxCommand*

Command to set connection options.

Parameters

- **project** (SpineToolboxProject) – project
- **connection** (LoggingConnection) – project
- **options** (*dict*) – containing options to be set

redo()

undo()

class spinetoolbox.project_commands.**AddSpecificationCommand**(*project*, *specification*, *save_to_disk*)

Bases: *SpineToolboxCommand*

Command to add item specification to a project.

Parameters

- **project** (ToolboxUI) – the toolbox
- **specification** (*ProjectItemSpecification*) – the spec
- **save_to_disk** (*bool*) – If True, save the specification to disk

redo()

undo()

class spinetoolbox.project_commands.**ReplaceSpecificationCommand**(*project*, *name*, *specification*)

Bases: *SpineToolboxCommand*

Command to replace item specification in project.

Parameters

- **project** (ToolboxUI) – the toolbox
- **name** (*str*) – the name of the spec to be replaced
- **specification** (*ProjectItemSpecification*) – the new spec

property is_critical

Returns True if this command needs to be undone before closing the project without saving changes.

redo()

undo()

class spinetoolbox.project_commands.**RemoveSpecificationCommand**(*project*, *name*)

Bases: *SpineToolboxCommand*

Command to remove specs from a project.

Parameters

- **project** (SpineToolboxProject) – the project
- **name** (*str*) – specification's name

redo()

undo()

class spinetoolbox.project_commands.**SaveSpecificationAsCommand**(*project*, *name*, *path*)

Bases: *SpineToolboxCommand*

Command to remove item specs from a project.

Parameters

- **project** (SpineToolboxProject) – the project
- **name** (*str*) – specification's name
- **path** (*str*) – new specification file location

redo()

undo()

spinetoolbox.project_item_icon

Classes for drawing graphics items on QGraphicsScene.

Module Contents

Classes

ProjectItemIcon	Base class for project item icons drawn in Design View.
ConnectorButton	Connector button graphics item. Used for Link drawing between project items.
ExecutionIcon	An icon to show information about the item's execution.
ExclamationIcon	An icon to notify that a ProjectItem is missing some configuration.
RankIcon	An icon to show the rank of a ProjectItem within its DAG.

class spinetoolbox.project_item_icon.**ProjectItemIcon**(*toolbox*, *icon_file*, *icon_color*)

Bases: PySide6.QtWidgets.QGraphicsPathItem

Base class for project item icons drawn in Design View.

Parameters

- **toolbox** (ToolboxUI) – QMainWindow instance
- **icon_file** (*str*) – Path to icon resource
- **icon_color** (*QColor*) – Icon's color

ITEM_EXTENT = 64

FONT_SIZE_PIXELS = 12

add_specification_icon(*spec_icon_path*)

Adds an SVG icon to bottom left corner of the item icon based on Tool Specification type.

Parameters
spec_icon_path (*str*) – Path to icon resource file.

remove_specification_icon()

Removes the specification icon SVG from the scene.

rect()

_update_path()

update_path(*rounded*)

_do_update_path(*rounded*)

finalize(*name*, *x*, *y*)

Names the icon and moves it by a given amount.

Parameters

- **name** (*str*) – icon's name

- **x** (*int*) – horizontal offset
- **y** (*int*) – vertical offset

_setup()

Sets up item attributes.

name()

Returns name of the item that is represented by this icon.

Returns
icon's name

Return type
str

update_name_item(*new_name*)

Sets a new text to name item.

Parameters
new_name (*str*) – icon's name

set_name_attributes()

Sets name item attributes (font, size, style, alignment).

_reposition_name_item()

Sets name item position (centered on top of the master icon).

conn_button(*position='left'*)

Returns item's connector button.

Parameters
position (*str*) – "left", "right" or "bottom"

Returns
connector button

Return type
QWidget

outgoing_connection_links()

Collects outgoing connection links.

Returns
outgoing links

Return type
list of LinkBase

incoming_links()

Collects incoming connection links.

Returns
outgoing links

Return type
list of LinkBase

_closest_connector(*pos*)

Returns the closest connector button to given scene pos.

_update_link_drawer_destination(*pos=None*)

Updates link drawer destination. If pos is None, then the link drawer would have no destination. Otherwise, the destination would be the connector button closest to pos.

hoverEnterEvent(*event*)

Sets a drop shadow effect to icon when mouse enters its boundaries.

Parameters
event (*QGraphicsSceneMouseEvent*) – Event

hoverMoveEvent(*event*)

hoverLeaveEvent(*event*)

Disables the drop shadow when mouse leaves icon boundaries.

Parameters
event (*QGraphicsSceneMouseEvent*) – Event

mousePressEvent(*event*)

Updates scene's icon group.

update_links_geometry()

Updates geometry of connected links to reflect this item's most recent position.

mouseReleaseEvent(*event*)

Clears pre-bump rects, and pushes a move icon command if necessary.

notify_item_move()

contextMenuEvent(*event*)

Show item context menu.

Parameters
event (*QGraphicsSceneMouseEvent*) – Mouse event

itemChange(*change*, *value*)

Reacts to item removal and position changes.

In particular, destroys the drop shadow effect when the item is removed from a scene and keeps track of item's movements on the scene.

Parameters

- **change** (*GraphicsItemChange*) – a flag signalling the type of the change
- **value** – a value related to the change

Returns
Whatever super() does with the value parameter

set_pos_without_bumping(*pos*)

Sets position without bumping other items. Needed for undoing move operations.

Parameters
pos (*QPointF*) –

_handle_collisions()

Handles collisions with other items.

make_room_for_item(*other*)

Makes room for another item.

Parameters
item (ProjectItemIcon) –

_restablish_bumped_items()

Moves bumped items back to their original position if no collision would happen anymore.

paint(*painter*, *option*, *widget=None*)

Sets a dashed pen if selected.

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

Bases: PySide6.QtWidgets.QGraphicsPathItem

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

Parameters

- **toolbox** (ToolboxUI) – QMainWindow instance
- **parent** (ProjectItemIcon) – parent graphics item
- **position** (*str*) – Either "top", "left", "bottom", or "right"

property parent

brush

hover_brush

rect()

update_path(*parent_radius*)

outgoing_links()

incoming_links()

parent_name()

Returns project item name owning this connector button.

project_item()

Returns the project item this connector button is attached to.

Returns
project item

Return type
ProjectItem

mousePressEvent(*event*)

Connector button mouse press event.

Parameters
event (*QGraphicsSceneMouseEvent*) – Event

_start_link(*event*)

set_friend_connectors_enabled(*enabled*)

Enables or disables all connectors in the parent.

This is called by LinkDrawer to disable invalid connectors while drawing and reenabling them back when done.

Parameters
enabled (*bool*) – True to enable connectors, False to disable

set_hover_brush()

set_normal_brush()

hoverEnterEvent(*event*)

Sets a darker shade to connector button when mouse enters its boundaries.

Parameters
event (*QGraphicsSceneMouseEvent*) – Event

hoverLeaveEvent(*event*)

Restore original brush when mouse leaves connector button boundaries.

Parameters
event (*QGraphicsSceneMouseEvent*) – Event

itemChange(*change*, *value*)

If this is being removed from the scene while it's the origin of the link drawer, put the latter to sleep.

class spinetoolbox.project_item_icon.**ExecutionIcon**(*parent*)

Bases: PySide6.QtWidgets.QGraphicsEllipseItem

An icon to show information about the item's execution.

Parameters
parent (ProjectItemIcon) – the parent item

_CHECK = '\uf00c'

_CROSS = '\uf00d'

_CLOCK = '\uf017'

_SKIP = '\uf054'

item_name()

_repaint(*text*, *color*)

mark_execution_waiting()

mark_execution_ignored()

mark_execution_started()

mark_execution_finished(*item_finish_state*)

hoverEnterEvent(*event*)

hoverLeaveEvent(*event*)

class spinetoolbox.project_item_icon.**ExclamationIcon**(*parent*)

Bases: PySide6.QtWidgets.QGraphicsTextItem

An icon to notify that a ProjectItem is missing some configuration.

Parameters
parent (ProjectItemIcon) – the parent item

FONT_SIZE_PIXELS = 14

clear_notifications()

Clear all notifications.

clear_other_notifications(*subtext*)

Remove notifications that don't include the given subtext.

add_notification(*text*)

Add a notification.

remove_notification(*subtext*)

Remove the first notification that includes given subtext.

hoverEnterEvent(*event*)

Shows notifications as tool tip.

Parameters
event (*QGraphicsSceneMouseEvent*) – Event

hoverLeaveEvent(*event*)

Hides tool tip.

Parameters
event (*QGraphicsSceneMouseEvent*) – Event

class spinetoolbox.project_item_icon.**RankIcon**(*parent*)

Bases: PySide6.QtWidgets.QGraphicsTextItem

An icon to show the rank of a ProjectItem within its DAG.

Parameters
parent (ProjectItemIcon) – the parent item

_make_path(*radius*)

update_path(*radius*)

set_rank(*rank*)

spinetoolbox.project_settings

Contains project-specific settings.

Module Contents

Classes

ProjectSettings	Spine Toolbox project settings.

class spinetoolbox.project_settings.**ProjectSettings**

Spine Toolbox project settings.

enable_execute_all: bool = True

to_dict()

Serializes the settings into a dictionary.

Returns
serialized settings

Return type
dict

static from_dict(*settings_dict*)

Deserializes settings from dictionary.

Parameters
settings_dict (*dict*) – serialized settings

Returns
deserialized settings

Return type
ProjectSettings

spinetoolbox.project_upgrader

Contains ProjectUpgrader class used in upgrading and converting projects and project dicts from earlier versions to the latest version.

Module Contents

Classes

ProjectUpgrader	Class to upgrade/convert projects from earlier versions to the current version.

Functions

_fix_1d_array_to_array(mappings)	Replaces '1d array' with 'array' for parameter type in Importer mappings.

class spinetoolbox.project_upgrader.**ProjectUpgrader**(*toolbox*)

Class to upgrade/convert projects from earlier versions to the current version.

Parameters
toolbox (ToolboxUI) – App main window instance

upgrade(*project_dict*, *project_dir*)

Upgrades the project described in given project dictionary to the latest version.

Parameters

- **project_dict** (*dict*) – Project configuration dictionary
- **project_dir** (*str*) – Path to current project directory

Returns
Latest version of the project info dictionary

Return type
dict

confirm_upgrade(*project_dir*)

Asks user whether to upgrade the project to a new version.

upgrade_to_latest(*v*, *project_dict*, *project_dir*)

Upgrades the given project dictionary to the latest version.

Parameters

- **v** (*int*) – Current version of the project dictionary
- **project_dict** (*dict*) – Project dictionary (JSON) to be upgraded
- **project_dir** (*str*) – Path to current project directory

Returns
Upgraded project dictionary

Return type
dict

static upgrade_v1_to_v2(*old*, *factories*)

Upgrades version 1 project dictionary to version 2.

Changes:
objects -> items, tool_specifications -> specifications store project item dicts under ["items"][<project item name>] instead of using their categories as keys specifications must be a dict instead of a list Add specifications["Tool"] that must be a dict Remove "short name" from all project items

Parameters

- **old** (*dict*) – Version 1 project dictionary
- **factories** (*dict*) – Mapping of item type to item factory

Returns
Version 2 project dictionary

Return type
dict

`upgrade_v2_to_v3`(*old*, *project_dir*, *factories*)

Upgrades version 2 project dictionary to version 3.

Changes:

1. Move "specifications" from "project" -> "Tool" to just "project"
2. The "mappings" from importer items are used to build Importer specifications

Parameters

- **old** (*dict*) – Version 2 project dictionary
- **project_dir** (*str*) – Path to current project directory
- **factories** (*dict*) – Mapping of item type to item factory

Returns
Version 3 project dictionary

Return type
dict

`static upgrade_v3_to_v4`(*old*)

Upgrades version 3 project dictionary to version 4.

Changes:

1. Rename "Exporter" item type to "GdxExporter"

Parameters
old (*dict*) – Version 3 project dictionary

Returns
Version 4 project dictionary

Return type
dict

`static upgrade_v4_to_v5`(*old*)

Upgrades version 4 project dictionary to version 5.

Changes:

1. Get rid of "Combiner" items.

Parameters
old (*dict*) – Version 4 project dictionary

Returns
Version 5 project dictionary

Return type
dict

static upgrade_v5_to_v6(*old*, *project_dir*)

Upgrades version 5 project dictionary to version 6.

Changes:

1. Data store URL labels do not have '{' and '}' anymore
2. Importer stores resource labels instead of serialized paths in "file_selection".
3. Gimlet's "selections" is now called "file_selection"
4. Gimlet stores resource labels instead of serialized paths in "file_selection".
5. Gimlet and Tool store command line arguments as serialized CmdLineArg objects, not serialized paths

Parameters

- **old** (*dict*) – Version 5 project dictionary
- **project_dir** (*str*) – Path to current project directory

Returns
Version 6 project dictionary

Return type
dict

static upgrade_v6_to_v7(*old*)

Upgrades version 6 project dictionary to version 7.

Changes:

1. Introduces Mergers in between DS -> DS links.

Parameters
old (*dict*) – Version 6 project dictionary

Returns
Version 7 project dictionary

Return type
dict

static upgrade_v7_to_v8(*old*)

Upgrades version 7 project dictionary to version 8.

Changes:

1. Move purge settings from items to their outgoing connections.

Parameters
old (*dict*) – Version 7 project dictionary

Returns
Version 8 project dictionary

Return type
dict

static upgrade_v8_to_v9(*old*)

Upgrades version 8 project dictionary to version 9.

Changes:

1. Remove ["project"]["name"] key

Parameters
old (*dict*) – Version 8 project dictionary

Returns
Version 9 project dictionary

Return type
dict

static upgrade_v9_to_v10(*old*)

Upgrades version 9 project dictionary to version 10.

Changes:

1. Remove connections from Gimlets and GDXExporters
2. Remove Gimlet items

Parameters
old (*dict*) – Version 9 project dictionary

Returns
Version 10 project dictionary

Return type
dict

static upgrade_v10_to_v11(*old*)

Upgrades version 10 project dictionary to version 11.

Changes:

1. Add ["project"]["settings"] key

Parameters
old (*dict*) – Version 10 project dictionary

Returns
Version 11 project dictionary

Return type
dict

static upgrade_v11_to_v12(*old*)

Upgrades version 11 project dictionary to version 12.

Changes:
1. Julia's execution settings are now Tool Spec settings instead of global settings Execution settings are local user settings so this only updates the project version to make sure that these projects cannot be opened with an older Toolbox version.

Parameters
old (*dict*) – Version 11 project dictionary

Returns
Version 12 project dictionary

Return type
dict

`static upgrade_v12_to_v13(old)`

Upgrades version 12 project dictionary to version 13.

Changes:
1. Connections now have enabled filter types field. Old projects should open just fine so this only updates the project version to make sure that these projects cannot be opened with an older Toolbox version.

Parameters
old (*dict*) – Version 12 project dictionary

Returns
Version 13 project dictionary

Return type
dict

`static make_unique_importer_specification_name(importer_name, label, k)`

`get_project_directory()`

Asks the user to select a new project directory. If the selected directory is already a Spine Toolbox project directory, asks if overwrite is ok. Used when opening a project from an old style project file (.proj).

Returns
Path to project directory or an empty string if operation is canceled.

Return type
str

`is_valid(v, p)`

Checks given project dict if it is valid for given version.

Parameters

- **v** (*int*) – project version to validate against
- **p** (*dict*) – project dictionary

Returns
True if project is valid, False otherwise

Return type
bool

`is_valid_v1(p)`

Checks that the given project JSON dictionary contains a valid version 1 Spine Toolbox project. Valid meaning, that it contains all required keys and values are of the correct type.

Parameters
p (*dict*) – Project information JSON

Returns
True if project is a valid version 1 project, False if it is not

Return type
bool

`is_valid_v2_to_v8`(*p*, *v*)

Checks that the given project JSON dictionary contains a valid version 2 to 8 Spine Toolbox project. Valid meaning, that it contains all required keys and values are of the correct type.

Parameters

- **p** (*dict*) – Project information JSON
- **v** (*int*) – Version

Returns
True if project is a valid version 2 to version 8 project, False if it is not

Return type
bool

`is_valid_v9_to_v10`(*p*)

Checks that the given project JSON dictionary contains a valid version 9 or 10 Spine Toolbox project. Valid meaning, that it contains all required keys and values are of the correct type.

Parameters
p (*dict*) – Project information JSON

Returns
True if project is a valid version 9 and 10 project, False otherwise

Return type
bool

`is_valid_v11_to_v12`(*p*)

Checks that the given project JSON dictionary contains a valid version 11 or 12 Spine Toolbox project. Valid meaning, that it contains all required keys and values are of the correct type.

Parameters
p (*dict*) – Project information JSON

Returns
True if project is a valid version 11 or 12 project, False otherwise

Return type
bool

`backup_project_file`(*project_dir*, *v*)

Makes a backup copy of project.json file.

`force_save`(*p*, *project_dir*)

Saves given project dictionary to project.json file. Used to force save project.json file when the project dictionary has been upgraded.

`spinetoolbox.project_upgrader._fix_1d_array_to_array`(*mappings*)

Replaces '1d array' with 'array' for parameter type in Importer mappings.

With spinedb_api >= 0.3, '1d array' parameter type was replaced by 'array'. Other settings in a mapping are backwards compatible except the name.

spinetoolbox.qthread_pool_executor

Qt-based thread pool executor.

Module Contents

Classes

_CustomQSemaphore	
QtBasedQueue	A Qt-based clone of queue.Queue.
QtBasedFuture	A Qt-based clone of concurrent.futures.Future.
QtBasedThread	A Qt-based clone of threading.Thread.
QtBasedThreadPoolExecutor	A Qt-based clone of concurrent.futures.ThreadPoolExecutor
SynchronousExecutor	

Functions

_set_future_result_and_exc(future, fn, *args, **kwargs)	

exception spinetoolbox.qthread_pool_executor.**TimeOutError**

Bases: Exception

An exception to raise when a timeouts expire

Initialize self. See help(type(self)) for accurate signature.

class spinetoolbox.qthread_pool_executor.**_CustomQSemaphore**

Bases: PySide6.QtCore.QSemaphore

tryAcquire(*n*, *timeout=None*)

class spinetoolbox.qthread_pool_executor.**QtBasedQueue**

A Qt-based clone of queue.Queue.

put(*item*)

get(*timeout=None*)

class spinetoolbox.qthread_pool_executor.**QtBasedFuture**

A Qt-based clone of concurrent.futures.Future.

set_result(*result*)

set_exception(*exc*)

result(*timeout=None*)

exception(*timeout=None*)

add_done_callback(*callback*)

class spinetoolbox.qthread_pool_executor.**QtBasedThread**(*target=None*, *args=()*)

Bases: PySide6.QtCore.QThread

A Qt-based clone of threading.Thread.

run()

class spinetoolbox.qthread_pool_executor.**QtBasedThreadPoolExecutor**(*max_workers=None*)

A Qt-based clone of concurrent.futures.ThreadPoolExecutor

submit(*fn*, **args*, ***kwargs*)

_spawn_thread()

_do_work()

shutdown()

class spinetoolbox.qthread_pool_executor.**SynchronousExecutor**

submit(*fn*, **args*, ***kwargs*)

shutdown()

spinetoolbox.qthread_pool_executor.**_set_future_result_and_exc**(*future*, *fn*, **args*, ***kwargs*)

spinetoolbox.spine_db_commands

QUndoCommand subclasses for modifying the db.

Module Contents

Classes

AgedUndoStack	
AgedUndoCommand	**param parent** The parent command, used for defining macros.
SpineDBCommand	Base class for all commands that modify a Spine DB.
AddItemsCommand	Base class for all commands that modify a Spine DB.
UpdateItemsCommand	Base class for all commands that modify a Spine DB.
AddUpdateItemsCommand	Base class for all commands that modify a Spine DB.
RemoveItemsCommand	Base class for all commands that modify a Spine DB.

class spinetoolbox.spine_db_commands.**AgedUndoStack**

Bases: PySide6.QtGui.QUndoStack

property redo_age

property undo_age

class spinetoolbox.spine_db_commands.**AgedUndoCommand**(*parent=None*, *identifier=-1*)

Bases: PySide6.QtGui.QUndoCommand

Parameters
parent (*QUndoCommand, optional*) – The parent command, used for defining macros.

property age

id()

override

ours()

mergeWith(*command*)

redo()

undo()

class spinetoolbox.spine_db_commands.**SpineDBCommand**(*db_mngr*, *db_map*, ***kwargs*)

Bases: *AgedUndoCommand*

Base class for all commands that modify a Spine DB.

Parameters

- **db_mngr** (SpineDBManager) – SpineDBManager instance
- **db_map** (*DiffDatabaseMapping*) – DiffDatabaseMapping instance

class spinetoolbox.spine_db_commands.**AddItemsCommand**(*db_mngr*, *db_map*, *item_type*, *data*, *check=True*, ***kwargs*)

Bases: *SpineDBCommand*

Base class for all commands that modify a Spine DB.

Parameters

- **db_mngr** (SpineDBManager) – SpineDBManager instance
- **db_map** (*DiffDatabaseMapping*) – DiffDatabaseMapping instance
- **data** (*list*) – list of dict-items to add
- **item_type** (*str*) – the item type

redo()

undo()

class spinetoolbox.spine_db_commands.**UpdateItemsCommand**(*db_mngr*, *db_map*, *item_type*, *data*, *check=True*, ***kwargs*)

Bases: *SpineDBCommand*

Base class for all commands that modify a Spine DB.

Parameters

- **db_mngr** (SpineDBManager) – SpineDBManager instance
- **db_map** (*DiffDatabaseMapping*) – DiffDatabaseMapping instance
- **item_type** (*str*) – the item type

- **data** (*list*) – list of dict-items to update

redo()

undo()

class spinetoolbox.spine_db_commands.**AddUpdateItemsCommand**(*db_mngr*, *db_map*, *item_type*, *data*, *check=True*, ***kwargs*)

Bases: *SpineDBCommand*

Base class for all commands that modify a Spine DB.

Parameters

- **db_mngr** (SpineDBManager) – SpineDBManager instance
- **db_map** (*DiffDatabaseMapping*) – DiffDatabaseMapping instance
- **item_type** (*str*) – the item type
- **data** (*list*) – list of dict-items to add-update

redo()

undo()

class spinetoolbox.spine_db_commands.**RemoveItemsCommand**(*db_mngr*, *db_map*, *item_type*, *ids*, *check=True*, ***kwargs*)

Bases: *SpineDBCommand*

Base class for all commands that modify a Spine DB.

Parameters

- **db_mngr** (SpineDBManager) – SpineDBManager instance
- **db_map** (*DiffDatabaseMapping*) – DiffDatabaseMapping instance
- **item_type** (*str*) – the item type
- **ids** (*set*) – set of ids to remove

redo()

undo()

spinetoolbox.spine_db_icon_manager

Provides SpineDBIconManager.

Module Contents

Classes

_SceneSvgRenderer	
SpineDBIconManager	A class to manage object_class icons for spine db editors.
SceneIconEngine	Specialization of QIconEngine used to draw scene-based icons.

Functions

_align_text_in_item(item)	
_center_scene(scene)	

spinetoolbox.spine_db_icon_manager.**_align_text_in_item**(*item*)

spinetoolbox.spine_db_icon_manager.**_center_scene**(*scene*)

class spinetoolbox.spine_db_icon_manager.**_SceneSvgRenderer**(*scene*)

Bases: PySide6.QtSvg.QSvgRenderer

class spinetoolbox.spine_db_icon_manager.**SpineDBIconManager**

A class to manage object_class icons for spine db editors.

update_icon_caches(*classes*)

Called after adding or updating entity classes. Stores display_icons and clears obsolete entries from the relationship class and entity group renderer caches.

_create_icon_renderer(*icon_code*, *color_code*)

icon_renderer(*icon_code*, *color_code*)

color_class_renderer(*entity_class*, *color_code*)

_create_class_renderer(*class_name*)

_create_multi_class_renderer(*dimension_name_list*)

class_renderer(*entity_class*)

multi_class_renderer(*dimension_name_list*)

_create_group_renderer(*class_name*)

group_renderer(*entity_class*)

static icon_from_renderer(*renderer*)

class spinetoolbox.spine_db_icon_manager.**SceneIconEngine**(*scene*)

Bases: *spinetoolbox.helpers.TransparentIconEngine*

Specialization of QIconEngine used to draw scene-based icons.

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

spinetoolbox.spine_db_manager

The SpineDBManager class.

Module Contents

Classes

SpineDBManager	Class to manage DBs within a project.

Functions

do_create_new_spine_database(url)	Creates a new spine database at the given url.

spinetoolbox.spine_db_manager.**do_create_new_spine_database**(*url*)

Creates a new spine database at the given url.

class spinetoolbox.spine_db_manager.**SpineDBManager**(*settings*, *parent*, *synchronous=False*)

Bases: PySide6.QtCore.QObject

Class to manage DBs within a project.

Parameters

- **settings** (*QSettings*) – Toolbox settings
- **parent** (*QObject, optional*) – parent object
- **synchronous** (*bool*) – If True, fetch database synchronously

property db_maps

property db_urls

error_msg

items_added

Emitted whenever items are added to a DB.

Parameters

- **str** – item type, such as "object_class"
- **dict** – mapping DatabaseMapping to list of added dict-items.

items_updated

Emitted whenever items are updated in a DB.

Parameters

- **str** – item type, such as "object_class"
- **dict** – mapping DatabaseMapping to list of updated dict-items.

items_removed

Emitted whenever items are removed from a DB.

Parameters

- **str** – item type, such as "object_class"
- **dict** – mapping DatabaseMapping to list of updated dict-items.

_connect_signals()

receive_error_msg(*db_map_error_log*)

receive_session_refreshed(*db_maps*)

receive_session_committed(*db_maps*, *cookie*)

receive_session_rolled_back(*db_maps*)

_get_worker(*db_map*)

Returns a worker.

Parameters
db_map (*DatabaseMapping*) – database mapping

Returns
worker for the db_map

Return type
SpineDBWorker

register_fetch_parent(*db_map*, *parent*)

Registers a fetch parent.

Parameters

- **db_map** (*DatabaseMapping*) – target database mapping
- **parent** (FetchParent) – fetch parent

can_fetch_more(*db_map*, *parent*)

Whether or not we can fetch more items of given type from given db.

Parameters

- **db_map** (*DatabaseMapping*) –
- **parent** (FetchParent) – The object that requests the fetching and that might want to react to further DB modifications.

Returns
bool

fetch_more(*db_map*, *parent*)

Fetches more items of given type from given db.

Parameters

- **db_map** (*DatabaseMapping*) –
- **parent** (FetchParent) – The object that requests the fetching.

get_icon_mngr(*db_map*)

Returns an icon manager for given db_map.

Parameters
db_map (*DatabaseMapping*) –

Returns
SpineDBIconManager

update_icons(*db_map*, *item_type*, *items*)

Runs when items are added or updated. Setups icons.

static db_map_key(*db_map*)

Creates an identifier for given db_map.

Parameters
db_map (*DatabaseMapping*) – database mapping

Returns
identification key

Return type
int

db_map_from_key(*key*)

Returns database mapping that corresponds to given identification key.

Parameters
key (*int*) – identification key

Returns
database mapping

Return type
DatabaseMapping

Raises
KeyError – raised if database map is not found

db_map(*url*)

Returns a database mapping for given URL.

Parameters
url (*str*) – a database URL

Returns
a database map or None if not found

Return type
DatabaseMapping

create_new_spine_database(*url*, *logger*, *overwrite=False*)

close_session(*url*)

Pops any db map on the given url and closes its connection.

Parameters
url (*str*) –

close_all_sessions()

Closes connections to all database mappings.

get_db_map(*url*, *logger*, *ignore_version_error=False*, *window=False*, *codename=None*, *create=False*)

Returns a DatabaseMapping instance from url if possible, None otherwise. If needed, asks the user to upgrade to the latest db version.

Parameters

- **url** (*str, URL*) –
- **logger** (LoggerInterface) –
- **ignore_version_error** (*bool, optional*) –
- **window** (*bool, optional*) –
- **codename** (*str, NoneType, optional*) –
- **create** (*bool, optional*) –

Returns

DatabaseMapping, NoneType

_do_get_db_map(*url*, ***kwargs*)

Returns a memorized DatabaseMapping instance from url. Called by *get_db_map*.

Parameters

- **url** (*str, URL*) –
- **codename** (*str, NoneType*) –
- **upgrade** (*bool*) –
- **create** (*bool*) –

Returns

DatabaseMapping

query(*db_map*, *sq_name*)

For tests.

add_db_map_listener(*db_map*, *listener*)

Adds listener for given db_map.

remove_db_map_listener(*db_map*, *listener*)

Removes db_map from the maps listener listens to.

db_map_listeners(*db_map*)

register_listener(*listener*, **db_maps*)

Register given listener for all given db_map's signals.

Parameters

- **listener** (*object*) –
- ***db_maps** –

unregister_listener(*listener*, **db_maps*, *dirty_db_maps=None*, *commit_dirty=False*, *commit_msg=''*)

Unregisters given listener from given db_map signals. If any of the db_maps becomes an orphan and is dirty, prompts user to commit or rollback.

Parameters

- **listener** (*object*) –
- ***db_maps** –

- **commit_dirty** (*bool*) – True to commit dirty database mapping, False to roll back
- **commit_msg** (*str*) – commit message

Returns
All the db maps that failed to commit

Return type
failed_db_maps (list)

add_data_store_db_map(*db_map*, *store*)

Adds a Data Store instance under a db_map into memory

remove_data_store_db_map(*db_map*, *store*)

Removes a Data Store instance from memory

update_data_store_db_maps()

Updates the db maps of the dict that maps db_maps to Data Stores

is_dirty(*db_map*)

Returns True if mapping has pending changes.

Parameters
db_map (*DatabaseMapping*) – database mapping

Returns
True if db_map has pending changes, False otherwise

Return type
bool

dirty(**db_maps*)

Filters clean mappings from given database maps.

Parameters
***db_maps** – mappings to check

Returns
dirty mappings

Return type
list of DatabaseMapping

dirty_and_without_editors(*listener*, **db_maps*)

Checks which of the given database mappings are dirty and have no editors.

Parameters

- **listener** (*Any*) – a listener object
- ***db_maps** – mappings to check

Returns
mappings that are dirty and don't have editors

Return type
list of DatabaseMapping

clean_up()

refresh_session(**db_maps*)

reset_session(**db_maps*)

commit_session(*commit_msg*, **dirty_db_maps*, *cookie=None*)

Commits the current session.

Parameters

- **commit_msg** (*str*) – commit message for all database maps
- ***dirty_db_maps** – dirty database maps to commit
- **cookie** (*object, optional*) – a free form identifier which will be forwarded to SpineDBWorker.commit_session

_do_commit_session(*db_map*, *commit_msg*, *cookie=None*)

Commits session for given db.

Parameters

- **db_map** (*DatabaseMapping*) – db map
- **commit_msg** (*str*) – commit message
- **cookie** (*Any*) – a cookie to include in receive_session_committed call

Returns
bool

notify_session_committed(*cookie*, **db_maps*)

Notifies manager and listeners when a commit has taken place by a third party.

Parameters

- **cookie** (*Any*) – commit cookie
- ***db_maps** – database maps that were committed

rollback_session(**dirty_db_maps*)

Rolls back the current session.

Parameters
***dirty_db_maps** – dirty database maps to commit

_do_rollback_session(*db_map*)

Rolls back session for given db.

Parameters
db_map (*DatabaseMapping*) – db map

entity_class_renderer(*db_map*, *entity_class_id*, *for_group=False*, *color=None*)

Returns an icon renderer for a given entity class.

Parameters

- **db_map** (*DatabaseMapping*) – database map
- **entity_class_id** (*int*) – entity class id
- **for_group** (*bool*) – if True, return the group object icon instead
- **color** (*QColor, optional*) – icon color

Returns
requested renderer or None if no entity class was found

Return type
QSvgRenderer

entity_class_icon(*db_map*, *entity_class_id*, *for_group=False*)

Returns an appropriate icon for a given entity class.

Parameters

- **db_map** (*DatabaseMapping*) – database map
- **entity_class_id** (*int*) – entity class' id
- **for_group** (*bool*) – if True, return the group object icon instead

Returns
requested icon or None if no entity class was found

Return type
QIcon

static get_item(*db_map*, *item_type*, *id_*)

Returns the item of the given type in the given db map that has the given id, or an empty dict if not found.

Parameters

- **db_map** (*DatabaseMapping*) –
- **item_type** (*str*) –
- **id** (*TempId*) –

Returns
the item

Return type
PublicItem

get_field(*db_map*, *item_type*, *id_*, *field*)

static get_items(*db_map*, *item_type*)

Returns a list of the items of the given type in the given db map.

Parameters

- **db_map** (*DatabaseMapping*) –
- **item_type** (*str*) –

Returns
list

get_items_by_field(*db_map*, *item_type*, *field*, *value*)

Returns a list of items of the given type in the given db map that have the given value for the given field.

Parameters

- **db_map** (*DatabaseMapping*) –
- **item_type** (*str*) –
- **field** (*str*) –
- **value** –

Returns
list

get_item_by_field(*db_map*, *item_type*, *field*, *value*)

Returns the first item of the given type in the given db map that has the given value for the given field Returns an empty dictionary if none found.

Parameters

- **db_map** (*DatabaseMapping*) –
- **item_type** (*str*) –
- **field** (*str*) –
- **value** –

Returns
dict

static display_data_from_parsed(*parsed_data*)

Returns the value's database representation formatted for Qt.ItemDataRole.DisplayRole.

static tool_tip_data_from_parsed(*parsed_data*)

Returns the value's database representation formatted for Qt.ItemDataRole.ToolTipRole.

_format_list_value(*db_map*, *item_type*, *value*, *list_value_id*)

get_value(*db_map*, *item*, *role=Qt.ItemDataRole.DisplayRole*)

Returns the value or default value of a parameter.

Parameters

- **db_map** (*DatabaseMapping*) – database mapping
- **item** (*PublicItem*) – parameter value item, parameter definition item, or list value item
- **role** (*Qt.ItemDataRole*) – data role

Returns
any

get_value_from_data(*data*, *role=Qt.ItemDataRole.DisplayRole*)

Returns the value or default value of a parameter directly from data. Used by `EmptyParameterModel.data()`.

Parameters

- **data** (*str*) – joined value and type
- **role** (*int, optional*) –

Returns
any

static _parse_value(*db_value*, *type_=None*)

_format_value(*parsed_value*, *role=Qt.ItemDataRole.DisplayRole*)

Formats the given value for the given role.

Parameters

- **parsed_value** (*object*) – A python object as returned by spinedb_api.from_database
- **role** (*int, optional*) –

get_value_indexes(*db_map*, *item_type*, *id_*)

Returns the value or default value indexes of a parameter.

Parameters

- **db_map** (*DatabaseMapping*) –
- **item_type** (*str*) – either "parameter_definition" or "parameter_value"
- **id** (*int*) – The parameter_value or definition id

get_value_index(*db_map*, *item_type*, *id_*, *index*, *role=Qt.ItemDataRole.DisplayRole*)

Returns the value or default value of a parameter for a given index.

Parameters

- **db_map** (*DatabaseMapping*) –
- **item_type** (*str*) – either "parameter_definition" or "parameter_value"
- **id** (*int*) – The parameter_value or definition id
- **index** – The index to retrieve
- **role** (*int, optional*) –

get_value_list_item(*db_map*, *id_*, *index*, *role=Qt.ItemDataRole.DisplayRole*)

Returns one value item of a parameter_value_list.

Parameters

- **db_map** (*DatabaseMapping*) –
- **id** (*int*) – The parameter_value_list id
- **index** (*int*) – The value item index
- **role** (*int, optional*) –

get_parameter_value_list(*db_map*, *id_*, *role=Qt.ItemDataRole.DisplayRole*)

Returns a parameter_value_list formatted for the given role.

Parameters

- **db_map** (*DatabaseMapping*) –
- **id** (*int*) – The parameter_value_list id
- **role** (*int, optional*) –

get_scenario_alternative_id_list(*db_map*, *scen_id*)

import_data(*db_map_data*, *command_text='Import data'*)

Imports the given data into given db maps using the dedicated import functions from spinedb_api. Condenses all in a single command for undo/redo.

Parameters

- **db_map_data** (*dict(DatabaseMapping, dict())*) – Maps dbs to data to be passed as keyword arguments to *get_data_for_import*
- **command_text** (*str, optional*) – What to call the command that condenses the operation.

add_alternatives(*db_map_data*)

Adds alternatives to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DatabaseMapping

add_scenarios(*db_map_data*)

Adds scenarios to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DatabaseMapping

add_entity_classes(*db_map_data*)

Adds entity classes to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DatabaseMapping

add_entities(*db_map_data*)

Adds entities to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DatabaseMapping

add_entity_groups(*db_map_data*)

Adds entity groups to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DatabaseMapping

add_entity_alternatives(*db_map_data*)

Adds entity alternatives to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DatabaseMapping

add_parameter_definitions(*db_map_data*)

Adds parameter definitions to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DatabaseMapping

add_parameter_values(*db_map_data*)

Adds parameter values to db without checking integrity.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DatabaseMapping

add_parameter_value_lists(*db_map_data*)

Adds parameter_value lists to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DatabaseMapping

add_list_values(*db_map_data*)

Adds parameter_value list values to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DatabaseMapping

add_metadata(*db_map_data*)

Adds metadata to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DatabaseMapping

add_entity_metadata(*db_map_data*)

Adds entity metadata to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DatabaseMapping

add_parameter_value_metadata(*db_map_data*)

Adds parameter value metadata to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DatabaseMapping

_add_ext_item_metadata(*db_map_data*, *item_type*)

add_ext_entity_metadata(*db_map_data*)

Adds entity metadata together with all necessary metadata to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DatabaseMapping

add_ext_parameter_value_metadata(*db_map_data*)

Adds parameter value metadata together with all necessary metadata to db.

Parameters
db_map_data (*dict*) – lists of items to add keyed by DatabaseMapping

update_alternatives(*db_map_data*)

Updates alternatives in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DatabaseMapping

update_scenarios(*db_map_data*)

Updates scenarios in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DatabaseMapping

update_entity_classes(*db_map_data*)

Updates entity classes in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DatabaseMapping

update_entities(*db_map_data*)

Updates entities in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DatabaseMapping

update_entity_alternatives(*db_map_data*)

Updates entity alternatives in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DatabaseMapping

update_parameter_definitions(*db_map_data*)

Updates parameter definitions in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DatabaseMapping

update_parameter_values(*db_map_data*)

Updates parameter values in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DatabaseMapping

update_expanded_parameter_values(*db_map_data*)

Updates expanded parameter values in db.

Parameters
db_map_data (*dict*) – lists of expanded items to update keyed by DatabaseMapping

update_parameter_value_lists(*db_map_data*)

Updates parameter_value lists in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DatabaseMapping

update_list_values(*db_map_data*)

Updates parameter_value list values in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DatabaseMapping

update_metadata(*db_map_data*)

Updates metadata in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DatabaseMapping

update_entity_metadata(*db_map_data*)

Updates entity metadata in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DatabaseMapping

update_parameter_value_metadata(*db_map_data*)

Updates parameter value metadata in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DatabaseMapping

_update_ext_item_metadata(*db_map_data*, *item_type*)

update_ext_entity_metadata(*db_map_data*)

Updates entity metadata in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DatabaseMapping

update_ext_parameter_value_metadata(*db_map_data*)

Updates parameter value metadata in db.

Parameters
db_map_data (*dict*) – lists of items to update keyed by DatabaseMapping

set_scenario_alternatives(*db_map_data*)

Sets scenario alternatives in db.

Parameters

db_map_data (*dict*) – lists of items to set keyed by DatabaseMapping

static get_data_to_set_scenario_alternatives(*db_map*, *scenarios*)

Returns data to add and remove, in order to set wide scenario alternatives.

Parameters

- **db_map** (*DatabaseMapping*) – the db_map
- **scenarios** – One or more wide scenario `dict` objects to set. Each item must include the following keys:
  - "id": integer scenario id
  - "alternative_id_list": list of alternative ids for that scenario

Returns

list: scenario_alternative `dict` objects to add. set: integer scenario_alternative ids to remove

purge_items(*db_map_item_types*, ***kwargs*)

Purges selected items from given database.

Parameters

- **db_map_item_types** (*dict*) – mapping from database map to list of purgable item types
- ****kwargs** – keyword arguments passed to `remove_items()`

add_items(*item_type*, *db_map_data*, *identifier=None*, ***kwargs*)

Pushes commands to add items to undo stack.

update_items(*item_type*, *db_map_data*, *identifier=None*, ***kwargs*)

Pushes commands to update items to undo stack.

add_update_items(*item_type*, *db_map_data*, *identifier=None*, ***kwargs*)

Pushes commands to add_update items to undo stack.

remove_items(*db_map_typed_ids*, *identifier=None*, ***kwargs*)

Pushes commands to remove items to undo stack.

get_command_identifier()

do_add_items(*db_map*, *item_type*, *data*, *check=True*)

do_update_items(*db_map*, *item_type*, *data*, *check=True*)

do_add_update_items(*db_map*, *item_type*, *data*, *check=True*)

do_remove_items(*db_map*, *item_type*, *ids*, *check=True*)

Removes items from database.

Parameters

- **db_map** – DatabaseMapping instance
- **item_type** (*str*) – database item type
- **ids** (*set*) – ids to remove

do_restore_items(*db_map*, *item_type*, *ids*)

Restores items in database.

Parameters

- **db_map** – DatabaseMapping instance
- **item_type** (*str*) – database item type
- **ids** (*set*) – ids to restore

static db_map_ids(*db_map_data*)

static db_map_class_ids(*db_map_data*)

find_cascading_entity_classes(*db_map_ids*)

Finds and returns cascading entity classes for the given dimension ids.

find_cascading_entities(*db_map_ids*)

Finds and returns cascading entities for the given element ids.

find_cascading_parameter_data(*db_map_ids*, *item_type*)

Finds and returns cascading parameter definitions or values for the given entity_class ids.

find_cascading_parameter_values_by_entity(*db_map_ids*)

Finds and returns cascading parameter values for the given entity ids.

find_cascading_parameter_values_by_definition(*db_map_ids*)

Finds and returns cascading parameter values for the given parameter_definition ids.

find_cascading_scenario_alternatives_by_scenario(*db_map_ids*)

Finds and returns cascading scenario alternatives for the given scenario ids.

find_groups_by_entity(*db_map_ids*)

Finds and returns groups for the given entity ids.

duplicate_scenario(*scen_data*, *dup_name*, *db_map*)

duplicate_entity(*orig_name*, *dup_name*, *class_name*, *db_maps*)

Duplicates entity, its parameter values and related multidimensional entities.

Parameters

- **orig_name** (*str*) – original entity's name
- **dup_name** (*str*) – duplicate's name
- **class_name** (*str*) – entity class name
- **db_maps** (*Iterable of DatabaseMapping*) – database mappings where duplication should take place

_get_data_for_export(*db_map_item_ids*)

export_data(*caller*, *db_map_item_ids*, *file_path*, *file_filter*)

_is_url_available(*url*, *logger*)

export_to_sqlite(*file_path*, *data_for_export*, *caller*)

Exports given data into SQLite file.

static export_to_json(*file_path*, *data_for_export*, *caller*)

Exports given data into JSON file.

static export_to_excel(*file_path*, *data_for_export*, *caller*)

Exports given data into Excel file.

get_items_for_commit(*db_map*, *commit_id*)

static get_all_multi_spine_db_editors()

Yields all instances of MultiSpineDBEditor currently open.

Yields
MultiSpineDBEditor

get_all_spine_db_editors()

Yields all instances of SpineDBEditor currently open.

Yields
SpineDBEditor

_get_existing_spine_db_editor(*db_url_codenames*)

open_db_editor(*db_url_codenames*, *reuse_existing_editor*)

Opens a SpineDBEditor with given urls. Uses an existing MultiSpineDBEditor if any. Also, if the same urls are open in an existing SpineDBEditor, just raises that one instead of creating another.

Parameters

- **db_url_codenames** (*dict*) – mapping url to codename
- **reuse_existing_editor** (*bool*) – if True and the same URL is already open, just raise the existing window

spinetoolbox.spine_db_parcel

SpineDBParcel class.

Module Contents

Classes

SpineDBParcel	A class to create parcels of data from a Spine db.

class spinetoolbox.spine_db_parcel.**SpineDBParcel**(*db_mngr*)

A class to create parcels of data from a Spine db. Mainly intended for the *Export selection* action in the Spine db editor:

- `push` methods push items with everything they need to live in a standalone db.
- `full_push` and `inner_push` methods do something more specific

Initializes the parcel object.

Parameters
db_mngr (SpineDBManager) –

property data

_get_field_values(*db_map*, *item_type*, *field*, *ids*)

Returns a list of field values for items of given type, having given ids.

push_entity_class_ids(*db_map_ids*)

Pushes entity_class ids.

push_entity_ids(*db_map_ids*)

Pushes entity ids.

push_parameter_value_list_ids(*db_map_ids*)

Pushes parameter_value_list ids.

push_parameter_definition_ids(*db_map_ids*)

Pushes parameter_definition ids.

push_parameter_value_ids(*db_map_ids*)

Pushes parameter_value ids.

push_entity_group_ids(*db_map_ids*)

Pushes entity group ids.

push_alternative_ids(*db_map_ids*)

Pushes alternative ids.

push_scenario_ids(*db_map_ids*)

Pushes scenario ids.

push_scenario_alternative_ids(*db_map_ids*)

Pushes scenario_alternative ids.

full_push_entity_class_ids(*db_map_ids*)

Pushes parameter definitions associated with given entity classes. This essentially full_pushes the entity classes, their parameter definitions, and their member entity classes.

full_push_entity_ids(*db_map_ids*)

Pushes parameter values associated with entities *and* their elements. This essentially full_pushes entities, all the parameter values, and all the necessary classes, definitions, and lists.

full_push_scenario_ids(*db_map_ids*)

inner_push_entity_ids(*db_map_ids*)

Pushes entity ids, cascading entity ids, and the associated parameter values, but not any entity classes or parameter definitions. Mainly intended for the *Duplicate entity* action.

inner_push_parameter_value_ids(*db_map_ids*)

Pushes parameter_value ids.

_update_ids(*db_map_ids*, *key*)

Updates ids for given database item.

Parameters

- **db_map_ids** (*dict*) – mapping from `DatabaseMapping` to ids or `Asterisk`
- **key** (*str*) – the key

_setdefault(*db_map*)

Adds new id sets for given db_map or returns existing ones.

Parameters
db_map (*DatabaseMapping*) – a database map

Returns
mapping from item name to set of ids

Return type
dict

spinetoolbox.spine_db_worker

The SpineDBWorker class.

Module Contents

Classes

SpineDBWorker	Does all the communication with a certain DB for SpineDBManager, in a non-GUI thread.

Attributes

_CHUNK_SIZE

spinetoolbox.spine_db_worker.**_CHUNK_SIZE = 10000**

class spinetoolbox.spine_db_worker.**SpineDBWorker**(*db_mngr*, *db_url*, *synchronous=False*)

Bases: PySide6.QtCore.QObject

Does all the communication with a certain DB for SpineDBManager, in a non-GUI thread.

_query_advanced

_get_parents(*item_type*)

clean_up()

get_db_map(**args*, ***kwargs*)

register_fetch_parent(*parent*)

Registers the given parent.

Parameters
parent (FetchParent) – parent to add

_iterate_mapping(*parent*)

Iterates the in-memory mapping for given parent while updating its `position` property. Iterated items are added to the parent if it accepts them.

Parameters
parent (FetchParent) – the parent.

Returns
Whether the parent can stop fetching from now

Return type
bool

can_fetch_more(*parent*)

Returns whether more data can be fetched for parent. Also, registers the parent to notify it of any relevant DB modifications later on.

Parameters
parent (FetchParent) – fetch parent

Returns
True if more data is available, False otherwise

Return type
bool

fetch_more(*parent*)

Fetches items from the database.

Parameters
parent (FetchParent) – fetch parent

_do_fetch_more(*parent*)

_fetch_more_later(*parents*)

_busy_db_map_fetch_more(*item_type*)

_handle_query_advanced(*item_type*, *chunk*)

_populate_commit_cache(*item_type*, *items*)

fetch_all()

close_db_map()

add_items(*item_type*, *orig_items*, *check*)

Adds items to db.

Parameters

- **item_type** (*str*) – item type
- **orig_items** (*list*) – dict-items to add
- **check** (*bool*) – Whether to check integrity

_wake_up_parents(*item_type*, *items*)

update_items(*item_type*, *orig_items*, *check*)

Updates items in db.

Parameters

- **item_type** (*str*) – item type
- **orig_items** (*list*) – dict-items to update
- **check** (*bool*) – Whether or not to check integrity

add_update_items(*item_type*, *orig_items*, *check*)

Add-updates items in db.

Parameters

- **item_type** (*str*) – item type
- **orig_items** (*list*) – dict-items to update
- **check** (*bool*) – Whether or not to check integrity

remove_items(*item_type*, *ids*, *check*)

Removes items from database.

Parameters

- **item_type** (*str*) – item type
- **ids** (*set*) – ids of items to remove

restore_items(*item_type*, *ids*)

Readds items to database.

Parameters

- **item_type** (*str*) – item type
- **ids** (*set*) – ids of items to restore

refresh_session()

Refreshes session.

reset_session()

Resets session.

spinetoolbox.spine_engine_manager

Contains SpineEngineManagerBase.

Module Contents

Classes

SpineEngineManagerBase	
LocalSpineEngineManager	
RemoteSpineEngineManager	Responsible for remote project execution.

Functions

make_engine_manager([remote_execution_enabled, job_id])	Returns either a Local or a remote Spine Engine Manager based on settings.

class spinetoolbox.spine_engine_manager.**SpineEngineManagerBase**

abstract run_engine(*engine_data*)

Runs an engine with given data.

Parameters
engine_data (*dict*) – The engine data.

abstract get_engine_event()

Gets next event from a running engine.

Returns
two element tuple: event type identifier string, and event data dictionary

Return type
tuple(str,dict)

abstract stop_engine()

Stops a running engine.

abstract answer_prompt(*prompter_id*, *answer*)

Answers prompt.

Parameters

- **prompter_id** (*int*) – The id of the prompter
- **answer** – The user's decision.

abstract restart_kernel(*connection_file*)

Restarts the jupyter kernel associated to given connection file.

Parameters
connection_file (*str*) – path of connection file

abstract shutdown_kernel(*connection_file*)

Shuts down the jupyter kernel associated to given connection file.

Parameters
connection_file (*str*) – path of connection file

abstract issue_persistent_command(*persistent_key*, *command*)

Issues a command to a persistent process.

Parameters

- **persistent_key** (*tuple*) – persistent identifier
- **command** (*str*) – command to issue

Returns
stdin, stdout, and stderr messages (dictionaries with two keys: type, and data)

Return type
generator

abstract is_persistent_command_complete(*persistent_key*, *command*)

Checks whether a command is complete.

Parameters

- **key** (*tuple*) – persistent identifier
- **cmd** (*str*) – command to issue

Returns
bool

abstract restart_persistent(*persistent_key*)

Restarts a persistent process.

Parameters
persistent_key (*tuple*) – persistent identifier

Returns
stdout and stderr messages (dictionaries with two keys: type, and data)

Return type
generator

abstract interrupt_persistent(*persistent_key*)

Interrupts a persistent process.

Parameters
persistent_key (*tuple*) – persistent identifier

abstract kill_persistent(*persistent_key*)

Kills a persistent process.

Parameters
persistent_key (*tuple*) – persistent identifier

abstract get_persistent_completions(*persistent_key*, *text*)

Returns a list of auto-completion options from given text.

Parameters

- **persistent_key** (*tuple*) – persistent identifier
- **text** (*str*) – text to complete

Returns
list of str

abstract get_persistent_history_item(*persistent_key*, *text*, *prefix*, *backwards*)

Returns an item from persistent history.

Parameters
persistent_key (*tuple*) – persistent identifier

Returns
history item or empty string if none

Return type
str

class spinetoolbox.spine_engine_manager.**LocalSpineEngineManager**

Bases: *SpineEngineManagerBase*

run_engine(*engine_data*)

Runs an engine with given data.

Parameters
engine_data (*dict*) – The engine data.

get_engine_event()

Gets next event from a running engine.

Returns
two element tuple: event type identifier string, and event data dictionary

Return type
tuple(str,dict)

stop_engine()

Stops a running engine.

answer_prompt(*prompter_id*, *answer*)

Answers prompt.

Parameters

- **prompter_id** (*int*) – The id of the prompter
- **answer** – The user's decision.

restart_kernel(*connection_file*)

Restarts the jupyter kernel associated to given connection file.

Parameters
connection_file (*str*) – path of connection file

shutdown_kernel(*connection_file*)

Shuts down the jupyter kernel associated to given connection file.

Parameters
connection_file (*str*) – path of connection file

kernel_managers()

issue_persistent_command(*persistent_key*, *command*)

Issues a command to a persistent process.

Parameters

- **persistent_key** (*tuple*) – persistent identifier
- **command** (*str*) – command to issue

Returns
stdin, stdout, and stderr messages (dictionaries with two keys: type, and data)

Return type
generator

is_persistent_command_complete(*persistent_key*, *command*)

Checks whether a command is complete.

Parameters

- **key** (*tuple*) – persistent identifier
- **cmd** (*str*) – command to issue

Returns
bool

`restart_persistent`(*persistent_key*)

Restarts a persistent process.

Parameters
persistent_key (*tuple*) – persistent identifier

Returns
stdout and stderr messages (dictionaries with two keys: type, and data)

Return type
generator

`interrupt_persistent`(*persistent_key*)

Interrupts a persistent process.

Parameters
persistent_key (*tuple*) – persistent identifier

`kill_persistent`(*persistent_key*)

Kills a persistent process.

Parameters
persistent_key (*tuple*) – persistent identifier

`get_persistent_completions`(*persistent_key*, *text*)

Returns a list of auto-completion options from given text.

Parameters

- **persistent_key** (*tuple*) – persistent identifier
- **text** (*str*) – text to complete

Returns
list of str

`get_persistent_history_item`(*persistent_key*, *text*, *prefix*, *backwards*)

Returns an item from persistent history.

Parameters
persistent_key (*tuple*) – persistent identifier

Returns
history item or empty string if none

Return type
str

`class spinetoolbox.spine_engine_manager.RemoteSpineEngineManager`(*job_id=''*)

Bases: *SpineEngineManagerBase*

Responsible for remote project execution.

Initializer.

`make_engine_client`(*host*, *port*, *security*, *sec_folder*, *ping=True*)

Creates a client for connecting to Spine Engine Server.

run_engine(*engine_data*)

Makes an engine client for communicating with the engine server. Starts a thread for monitoring the DAG execution on server.

Parameters
engine_data (*dict*) – The engine data.

get_engine_event()

Returns the next engine execution event.

clean_up()

Closes EngineClient and joins _runner thread if still active.

stop_engine()

Sends a request to stop execution on Server then waits for _runner thread to end.

_run()

Sends a start execution request to server with the job Id. Sets up a subscribe socket according to the publish port received from server. Passes received events to SpineEngineWorker for processing. After execution has finished, downloads new files from server.

answer_prompt(*prompter_id*, *answer*)

See base class.

restart_kernel(*connection_file*)

See base class.

shutdown_kernel(*connection_file*)

See base class.

is_persistent_command_complete(*persistent_key*, *command*)

Checks whether a command is complete.

Parameters

- **key** (*tuple*) – persistent identifier
- **cmd** (*str*) – command to issue

Returns
bool

issue_persistent_command(*persistent_key*, *command*)

Issues a command to a persistent process.

Parameters

- **persistent_key** (*tuple*) – persistent identifier
- **command** (*str*) – command to issue

Returns
stdin, stdout, and stderr messages (dictionaries with two keys: type, and data)

Return type
generator

restart_persistent(*persistent_key*)

See base class.

interrupt_persistent(*persistent_key*)

See base class.

kill_persistent(*persistent_key*)

See base class.

get_persistent_completions(*persistent_key*, *text*)

See base class.

get_persistent_history_item(*persistent_key*, *text*, *prefix*, *backwards*)

Returns an item from persistent history.

Parameters
persistent_key (*tuple*) – persistent identifier

Returns
history item or empty string if none

Return type
str

spinetoolbox.spine_engine_manager.**make_engine_manager**(*remote_execution_enabled=False*, *job_id=''*)

Returns either a Local or a remote Spine Engine Manager based on settings.

Parameters

- **remote_execution_enabled** (*bool*) – True returns a local Spine Engine Manager instance,
- **instance** (*False returns a remote Spine Engine Manager*) –
- **job_id** (*str*) – Server execution job Id

spinetoolbox.spine_engine_worker

Contains SpineEngineWorker.

Module Contents

Classes

SpineEngineWorker

param engine_data
engine data

Functions

_handle_dag_execution_started(project_items)	
_handle_node_execution_ignored(project_items)	
_handle_node_execution_started(item, direction)	
_handle_node_execution_finished(item, direction, ...)	
_handle_event_message_arrived(item, filter_id, ...)	
_handle_process_message_arrived(item, filter_id, ...)	
_handle_prompt_arrived(prompt, engine_mngr[, logger])	
_handle_flash_arrived(connection)	
_mark_all_items_failed(items)	Fails all project items.

spinetoolbox.spine_engine_worker.**_handle_dag_execution_started**(*project_items*)

spinetoolbox.spine_engine_worker.**_handle_node_execution_ignored**(*project_items*)

spinetoolbox.spine_engine_worker.**_handle_node_execution_started**(*item*, *direction*)

spinetoolbox.spine_engine_worker.**_handle_node_execution_finished**(*item*, *direction*, *item_state*)

spinetoolbox.spine_engine_worker.**_handle_event_message_arrived**(*item*, *filter_id*, *msg_type*, *msg_text*)

spinetoolbox.spine_engine_worker.**_handle_process_message_arrived**(*item*, *filter_id*, *msg_type*, *msg_text*)

spinetoolbox.spine_engine_worker.**_handle_prompt_arrived**(*prompt*, *engine_mngr*, *logger=None*)

spinetoolbox.spine_engine_worker.**_handle_flash_arrived**(*connection*)

spinetoolbox.spine_engine_worker.**_mark_all_items_failed**(*items*)

Fails all project items.

Parameters

items (*list of ProjectItem*) – project items

class spinetoolbox.spine_engine_worker.**SpineEngineWorker**(*engine_data*, *dag*, *dag_identifier*, *project_items*, *connections*, *logger*, *job_id*)

Bases: PySide6.QtCore.QObject

Parameters

- **engine_data** (*dict*) – engine data
- **dag** (*DirectedGraphHandler*) –
- **dag_identifier** (*str*) –

- **project_items** (*dict*) – mapping from project item name to ProjectItem
- **connections** (*dict*) – mapping from jump name to LoggingConnection or LoggingJump
- **logger** (LoggerInterface) – a logger
- **job_id** (*str*) – Job id for remote execution

property job_id

property engine_data

Engine data dictionary.

finished

_mark_items_ignored

_dag_execution_started

_node_execution_started

_node_execution_finished

_event_message_arrived

_process_message_arrived

_prompt_arrived

_flash_arrived

_all_items_failed

get_engine_data()

Returns the engine data. Together with self.set_engine_data() it can be used to modify the workflow after it's initially created. We use it at the moment for creating Julia sysimages.

Returns
dict

set_engine_data(*engine_data*)

Sets the engine data.

Parameters
engine_data (*dict*) – New data

_handle_event_message_arrived_silent(*item*, *filter_id*, *msg_type*, *msg_text*)

_handle_process_message_arrived_silent(*item*, *filter_id*, *msg_type*, *msg_text*)

stop_engine()

engine_final_state()

thread()

_connect_log_signals(*silent*)

start(*silent=False*)

Connects log signals.

Parameters

silent (*bool, optional*) – If True, log messages are not forwarded to the loggers but saved in internal dicts.

_included_and_ignored_items()

Returns two lists, where the first one contains project items that are about to be executed and the second one contains project items that are about to be ignored.

_included_items(*permitted_items*, *connections*)

Collects a list of project item names that are going to be executed in this DAG based on execution permits and connections in the DAG.

Parameters

- **permitted_items** (*dict*) – Mapping of item names to bool. True items have been selected by user for execution.
- **connections** (*list*) – Serialized connections

Returns

Project item names

Return type

list

do_work()

Does the work and emits finished when done.

_process_event(*event_type*, *data*)

_handle_prompt(*prompt*)

_handle_flash(*flash*)

_handle_standard_execution_msg(*msg*)

_handle_persistent_execution_msg(*msg*)

_handle_kernel_execution_msg(*msg*)

_handle_process_msg(*data*)

_do_handle_process_msg(*item_name*, *filter_id*, *msg_type*, *msg_text*)

_handle_event_msg(*data*)

_do_handle_event_msg(*item_name*, *filter_id*, *msg_type*, *msg_text*)

_handle_node_execution_started(*data*)

_do_handle_node_execution_started(*item_name*, *direction*)

Starts item icon animation when executing forward.

_handle_node_execution_finished(*data*)

_do_handle_node_execution_finished(*item_name*, *direction*, *state*, *item_state*)

_handle_server_status_msg(*data*)

clean_up()

spinetoolbox.ui_main

Contains a class for the main window of Spine Toolbox.

Module Contents

Classes

ToolboxUI	Class for application main GUI functions.

class spinetoolbox.ui_main.**ToolboxUI**

Bases: PySide6.QtWidgets.QMainWindow

Class for application main GUI functions.

Initializes application and main window.

msg

msg_success

msg_error

msg_warning

msg_proc

msg_proc_error

information_box

error_box

jupyter_console_requested

kernel_shutdown

persistent_console_requested

eventFilter(*obj*, *ev*)

_setup_properties_title()

connect_signals()

Connect signals.

static set_error_mode()

Sets Windows error mode to show all error dialog boxes from subprocesses.

See https://docs.microsoft.com/en-us/windows/win32/api/errhandlingapi/nf-errhandlingapi-seterrormode for documentation.

_update_qsettings()

Updates obsolete settings.

_update_execute_enabled()

_update_execute_selected_enabled()

Enables or disables execute selected action based on the number of selected items.

update_window_modified(*clean*)

Updates window modified status and save actions depending on the state of the undo stack.

parse_project_item_modules()

Collects data from project item factories.

set_work_directory(*new_work_dir=None*)

Creates a work directory if it does not exist or changes the current work directory to given.

Parameters
new_work_dir (*str, optional*) – If given, changes the work directory to given and creates the directory if it does not exist.

project()

Returns current project or None if no project open.

Returns
current project or None

Return type
SpineToolboxProject

qsettings()

Returns application preferences object.

item_specification_factories()

Returns project item specification factories.

Returns
specification factories

Return type
list of ProjectItemSpecificationFactory

update_window_title()

Updates main window title.

init_project(*project_dir*)

Initializes project at application start-up.

Opens the last project that was open when app was closed (if enabled in Settings) or starts the app without a project.

Parameters
project_dir (*str*) – project directory

new_project()

Opens a file dialog where user can select a directory where a project is created. Pops up a question box if selected directory is not empty or if it already contains a Spine Toolbox project. Initial project name is the directory name.

create_project(*proj_dir*)

Creates new project and sets it active.

Parameters
proj_dir (*str*) – Path to project directory

open_project(*load_dir=None*)

Opens project from a selected or given directory.

Parameters
load_dir (*str, optional*) – Path to project base directory. If default value is used, a file explorer dialog is opened where the user can select the project to open.

Returns
True when opening the project succeeded, False otherwise

Return type
bool

restore_project(*project_dir*, *ask_confirmation=True*)

Initializes UI, Creates project, models, connections, etc., when opening a project.

Parameters

- project_dir (*str*) – Project directory
- ask_confirmation (*bool*) – True closes the previous project with a confirmation box if user has enabled this

Returns
True when restoring project succeeded, False otherwise

Return type
bool

_toolbars()

Yields all toolbars in the window.

set_toolbar_colored_icons(*checked*)

_disable_project_actions()

Disables all project-related actions, except New project, Open project and Open recent. Called in the constructor and when closing a project.

_enable_project_actions()

Enables all project-related actions. Called when a new project is created and when a project is opened.

refresh_toolbars()

Set toolbars' color using the highest possible contrast.

show_recent_projects_menu()

Updates and sets up the recent projects menu to File-Open recent menu item.

fetch_kernels()

Starts a thread for fetching local kernels.

stop_fetching_kernels()

Terminates kernel fetcher thread.

restore_override_cursor()

Restores default mouse cursor.

save_project()

Saves project.

save_project_as()

Asks user for a new project directory and duplicates the current project there. The name of the duplicated project will be the new directory name. The duplicated project is activated.

close_project(*ask_confirmation=True*, *clear_event_log=True*)

Closes the current project.

Parameters

- **ask_confirmation** (*bool*) – if False, no confirmation whatsoever is asked from user
- **clear_event_log** (*bool*) – if True, the event log is cleared after closing the project

Returns
True when no project open or when it's closed successfully, False otherwise.

Return type
bool

set_project_description(*_=False*)

Opens a dialog where the user can enter a new description for the project.

init_specification_model()

Initializes specification model.

make_item_properties_uis()

_make_properties_tab(*properties_ui*)

add_project_items(*items_dict*)

Pushes an AddProjectItemsCommand to the undo stack.

Parameters
items_dict (*dict*) – mapping from item name to item dictionary

supports_specifications(*item_type*)

Returns True if given project item type supports specifications.

Returns
True if item supports specifications, False otherwise

Return type
bool

restore_ui()

Restore UI state from previous session.

clear_ui()

Clean UI to make room for a new or opened project.

undo_critical_commands()

Undoes critical commands in the undo stack.

Returns
False if any critical commands aren't successfully undone

Return type
Bool

overwrite_check(*project_dir*)

Checks if given directory is a project directory and/or empty And asks the user what to do in that case.

Parameters
project_dir (*str*) – Abs. path to a directory

Returns
True if user wants to overwrite an existing project or if the directory is not empty and the user wants to make it into a Spine Toolbox project directory anyway. False if user cancels the action.

Return type
bool

refresh_active_elements(*active_project_item*, *active_link_item*, *selected_item_names*)

_activate_properties_tab()

_set_active_project_item(*active_project_item*)

Activates given project item.

Parameters
active_project_item (*ProjectItemBase or NoneType*) – Active project item

_set_active_link_item(*active_link_item*)

Activates given link and connects it to the corresponding Properties widget.

Parameters
active_link_item (LoggingConnection *or* LoggingJump, *optional*) – Active link

activate_no_selection_tab()

Shows 'No Selection' tab.

activate_item_tab()

Shows active project item properties tab according to item type.

activate_link_tab()

Shows link properties tab.

update_properties_ui()

_get_active_properties_widget()

Returns the active item's or link's properties widget or None if no item or link is active.

add_specification(*specification*)

Pushes an AddSpecificationCommand to undo stack.

import_specification()

Opens a file dialog where the user can select an existing specification definition file (.json). If file is valid, pushes AddSpecificationCommand to undo stack.

replace_specification(*name*, *specification*)

Pushes an ReplaceSpecificationCommand to undo stack.

repair_specification(*name*)

Repairs specification if it is broken.

Parameters
name (*str*) – Specification's name

prompt_save_location(*title*, *proposed_path*, *file_filter*)

Shows a dialog for the user to select a path to save a file.

Parameters

- **title** (*str*) – Dialog window title
- **proposed_path** (*str*) – Proposed location.
- **file_filter** (*str*) – File extension filter

Returns
Absolute path or None if dialog was cancelled

Return type
str

_log_specification_saved(*name*, *path*)

Prints a message in the event log, saying that given spec was saved in a certain location, together with a clickable link to change the location.

Parameters

- **name** (*str*) – Specification's name
- **path** (*str*) – Specification's file path

remove_all_items()

Pushes a RemoveAllProjectItemsCommand to the undo stack.

register_anchor_callback(*url*, *callback*)

Registers a callback for a given anchor in event log, see `open_anchor()`. Used by `ToolFactory.repair_specification()`.

Parameters

- **url** (*str*) – Anchor url
- **callback** (*function*) – Function to call when the anchor is clicked

open_anchor(*qurl*)

Open file explorer in the directory given in qurl.

Parameters
qurl (*QUrl*) – The url to open

_change_specification_file_location(*name*)

Prompts user for new location for a project item specification.

Delegates saving to project if one is open by pushing a command to the undo stack, otherwise tries to find the specification from the plugin manager.

Parameters
name (*str*) – Specification's name

show_specification_context_menu(*ind*, *global_pos*)

Context menu for item specifications.

Parameters

- **ind** (*QModelIndex*) – In the ProjectItemSpecificationModel
- **global_pos** (*QPoint*) – Mouse position

edit_specification(*index*, *item*)

Opens a specification editor widget.

Parameters

- **index** (*QModelIndex*) – Index of the item (from double-click or context menu signal)
- **item** (ProjectItem, *optional*) –

remove_specification(*index*)

Removes specification from project.

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

open_specification_file(*index*)

Open the specification definition file in the default (.json) text-editor.

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

new_db_editor()

_handle_zoom_minus_pressed()

Slot for handling case when '-' button in menu is pressed.

_handle_zoom_plus_pressed()

Slot for handling case when '+' button in menu is pressed.

_handle_zoom_reset_pressed()

Slot for handling case when 'reset zoom' button in menu is pressed.

add_zoom_action()

Setups zoom widget action in view menu.

restore_dock_widgets()

Dock all floating and or hidden QDockWidgets back to the main window.

_add_execute_actions()

Adds execution handler actions to the main window.

set_debug_qactions()

Sets shortcuts for QActions that may be needed in debugging.

add_menu_actions()

Adds extra actions to Edit and View menu.

toggle_properties_tabbar_visibility()

Shows or hides the tab bar in properties dock widget. For debugging purposes.

update_datetime()

Returns a boolean, which determines whether date and time is prepended to every Event Log message.

add_message(*msg*)

Appends a regular message to the Event Log.

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

add_success_message(*msg*)

Appends a message with green text to the Event Log.

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

add_error_message(*msg*)

Appends a message with red color to the Event Log.

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

add_warning_message(*msg*)

Appends a message with yellow (golden) color to the Event Log.

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

add_process_message(*msg*)

Writes message from stdout to the Event Log.

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

add_process_error_message(*msg*)

Writes message from stderr to the Event Log.

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

override_console_and_execution_list()

_override_console()

Sets the jupyter console of the active project item in Jupyter Console and updates title.

_do_override_console(*console*)

_override_execution_list()

Displays executions of the active project item in Executions and updates title.

_restore_original_console()

Sets the Console back to the original.

_set_override_console(*console*)

_refresh_console_execution_list()

Refreshes console executions as the active project item starts new executions.

_select_console_execution(*current*, *_previous*)

Sets the console of the selected execution in Console.

show_add_project_item_form(*item_type*, *x=0*, *y=0*, *spec=''*)

Show add project item widget.

supports_specification(*item_type*)

Returns True if given item type supports specifications.

Parameters
item_type (*str*) – Item's type

Returns
True if item supports specifications, False otherwise

Return type
bool

show_specification_form(*item_type*, *specification=None*, *item=None*, ***kwargs*)

Shows specification widget.

Parameters

- **item_type** (*str*) – Item's type
- **specification** (*ProjectItemSpecification, optional*) – Specification
- **item** (ProjectItem, *optional*) – Project item
- ****kwargs** – Parameters passed to the specification widget

static get_all_multi_tab_spec_editors(*item_type*)

_get_existing_spec_editor(*item_type*, *specification*, *item*)

show_settings()

Shows the Settings widget.

show_about()

Shows the About Spine Toolbox widget.

show_user_guide()

Opens Spine Toolbox documentation index page in browser.

show_getting_started_guide()

Opens Spine Toolbox Getting Started HTML page in browser.

retrieve_project()

Retrieves project from server.

engine_server_settings()

Returns the user given Spine Engine Server settings in a tuple.

show_project_or_item_context_menu(*pos*, *item*)

Creates and shows the project item context menu.

Parameters

- **pos** (*QPoint*) – Mouse position
- **item** (ProjectItem, *optional*) – Project item or None

show_link_context_menu(*pos*, *link*)

Shows the Context menu for connection links.

Parameters

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

refresh_edit_action_states()

Sets the enabled/disabled state for copy, paste, duplicate, remove and rename actions in File-Edit menu, project tree view context menu, and in Design View context menus just before the menus are shown to user.

`disable_edit_actions()`

Disables edit actions.

`enable_edit_actions()`

Enables project item edit actions after a QMenu has been shown. This is needed to enable keyboard shortcuts (e.g. Ctrl-C & del) again.

`_tasks_before_exit()`

Returns a list of tasks to perform before exiting the application.

Possible tasks are:

- *"prompt exit"*: prompt user if quitting is really desired
- *"prompt save"*: prompt user if project should be saved before quitting
- *"save"*: save project before quitting

Returns
Zero or more tasks in a list

Return type
list

`_perform_pre_exit_tasks()`

Prompts user to confirm quitting and saves the project if necessary.

Returns
True if exit should proceed, False if the process was cancelled

Return type
bool

`_confirm_exit()`

Confirms exiting from user.

Returns
True if exit should proceed, False if user cancelled

Return type
bool

`_confirm_project_close()`

Confirms exit from user and saves the project if requested.

Returns
True if exiting should proceed, False if user cancelled

Return type
bool

`remove_path_from_recent_projects(p)`

Removes entry that contains given path from the recent project files list in QSettings.

Parameters
p (*str*) – Full path to a project directory

`clear_recent_projects()`

Clears recent projects list in File->Open recent menu.

update_recent_projects()

Adds a new entry to QSettings variable that remembers twenty most recent project paths.

closeEvent(*event*)

Method for handling application exit.

Parameters
event (*QCloseEvent*) – PySide6 event

_serialize_selected_items()

Serializes selected project items into a dictionary.

The serialization protocol tries to imitate the format in which projects are saved.

Returns
a dict containing serialized version of selected project items

Return type
dict

_deserialized_item_position_shifts(*item_dicts*)

Calculates horizontal and vertical shifts for project items being deserialized.

If the mouse cursor is on the Design view we try to place the items unders the cursor. Otherwise, the items will get a small shift, so they don't overlap a possible item below. In case the items don't fit the scene rect we clamp their coordinates within it.

Parameters
item_dicts (*dict*) – Dictionary of serialized items being deserialized

Returns
a tuple of (horizontal shift, vertical shift) in scene's coordinates

Return type
tuple

static _set_deserialized_item_position(*item_dict*, *shift_x*, *shift_y*, *scene_rect*)

Moves item's position by shift_x and shift_y while keeping it within the limits of scene_rect.

_deserialize_items(*items_dict*, *duplicate_files=False*)

Deserializes project items from a dictionary and adds them to the current project.

Parameters
items_dict (*dict*) – Serialized project items

project_item_to_clipboard()

Copies the selected project items to system's clipboard.

project_item_from_clipboard(*duplicate_files=False*)

Adds project items in system's clipboard to the current project.

Parameters
duplicate_files (*bool*) – Duplicate files boolean

duplicate_project_item(*duplicate_files=False*)

Duplicates the selected project items.

_add_item_edit_actions()

Adds generic actions to Design View.

_show_message_box(*title*, *message*)

Shows an information message box.

_show_error_box(*title*, *message*)

Shows an error message with the given title and message.

_connect_project_signals()

Connects signals emitted by project.

_execute_project(*_=False*)

Executes all DAGs in project.

_execute_selection(*_=False*)

Executes selected items.

_stop_execution(*_=False*)

Stops execution in progress.

_set_execution_in_progress()

_unset_execution_in_progress()

set_icon_and_properties_ui(*item_name*)

Adds properties UI to given project item.

Parameters
item_name (*str*) – Item's name

project_item_properties_ui(*item_type*)

Returns the properties tab widget's ui.

Parameters
item_type (*str*) – Project item's type

Returns
Item's properties tab widget

Return type
QWidget

project_item_icon(*item_type*)

_open_project_directory(*_*)

Opens project's root directory in system's file browser.

_open_project_item_directory(*_*)

Opens active project item's directory in system's file browser.

_remove_selected_items(*_*)

Pushes commands to remove selected project items and links from project.

_rename_project_item(*_*)

Renames active project item.

project_item_context_menu(*additional_actions*)

Creates a context menu for project items.

Parameters
additional_actions (*list of QAction*) – Actions to be prepended to the menu

Returns
Project item context menu

Return type
QMenu

`start_detached_jupyter_console`(*kernel_name*, *icon*, *conda*)

Launches a new detached Console with the given kernel name or activates an existing Console if the kernel is already running.

Parameters

- **kernel_name** (*str*) – Requested kernel name
- **icon** (*QIcon*) – Icon representing the kernel language
- **conda** (*bool*) – Is this a Conda kernel?

`_setup_jupyter_console`(*item*, *filter_id*, *kernel_name*, *connection_file*, *connection_file_dict*)

Sets up jupyter console, eventually for a filter execution.

Parameters

- **item** (ProjectItem) – Item
- **filter_id** (*str*) – Filter identifier
- **kernel_name** (*str*) – Jupyter kernel name
- **connection_file** (*str*) – Path to connection file
- **connection_file_dict** (*dict*) – Contents of connection file when kernel manager runs on Spine Engine Server

`_handle_kernel_shutdown`(*item*, *filter_id*)

Closes the kernel client when kernel manager has been shutdown due to an enabled 'Kill consoles at the end of execution' option.

Parameters

- **item** (ProjectItem) – Item
- **filter_id** (*str*) – Filter identifier

`_setup_persistent_console`(*item*, *filter_id*, *key*, *language*)

Sets up persistent console, eventually for a filter execution.

Parameters

- **item** (ProjectItem) – Item
- **filter_id** (*str*) – Filter identifier
- **key** (*tuple*) – Key
- **language** (*str*) – Language (e.g. 'python' or 'julia')

`persistent_killed`(*item*, *filter_id*)

`add_persistent_stdin`(*item*, *filter_id*, *data*)

`add_persistent_stdout`(*item*, *filter_id*, *data*)

`add_persistent_stderr`(*item*, *filter_id*, *data*)

_get_console(*item*, *filter_id*)

_make_jupyter_console(*item*, *kernel_name*, *connection_file*)

Creates a new JupyterConsoleWidget for given connection file if none exists yet, and returns it.

Parameters

- **item** (ProjectItem) – Item that owns the console
- **kernel_name** (*str*) – Name of the kernel
- **connection_file** (*str*) – Path of kernel connection file

Returns

JupyterConsoleWidget

_make_persistent_console(*item*, *key*, *language*)

Creates a new PersistentConsoleWidget for given process key.

Parameters

- **item** (ProjectItem) – Item that owns the console
- **key** (*tuple*) – persistent process key in spine engine
- **language** (*str*) – for syntax highlighting and prompting, etc.

Returns

PersistentConsoleWidget

_cleanup_jupyter_console(*conn_file*)

Removes reference to a Jupyter Console and closes the kernel manager on Engine.

_shutdown_engine_kernels()

Shuts down all persistent and Jupyter kernels managed by Spine Engine.

_close_consoles()

Closes all Persistent and Jupyter Console widgets.

restore_and_activate()

Brings the app main window into focus.

static _make_log_entry_title(*title*)

make_execution_timestamp(*timestamp*)

Appends a timestamp to Event Log.

Parameters

timestamp (*str*) – Time stamp

add_log_message(*item_name*, *filter_id*, *message*)

Adds a message to an item's execution log.

Parameters

- **item_name** (*str*) – item name
- **filter_id** (*str*) – filter identifier
- **message** (*str*) – formatted message

spinetoolbox.version

Version info for Spine Toolbox package. Inspired by python sys.version and sys.version_info.

Module Contents

Classes

VersionInfo	A class for a named tuple containing the five components of the version number: major, minor,

Attributes

split
__version_info__
__version__

class spinetoolbox.version.**VersionInfo**

Bases: NamedTuple

A class for a named tuple containing the five components of the version number: major, minor, micro, releaselevel, and serial. All values except releaselevel are integers; the release level is 'dev', 'alpha', 'beta', 'candidate', or 'final'.

major: int

minor: int

micro: int

releaselevel: str

serial: int

__str__() → str

Create a version string following PEP 440

spinetoolbox.version.**split**

spinetoolbox.version.**__version_info__**

spinetoolbox.version.**__version__**

20.1.3 Package Contents

spinetoolbox.**__version__**

spinetoolbox.**__version_info__**

CHAPTER

TWENTYONE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[CB14] Chris Beams. 2014. 'How to Write a Git Commit Message.' https://chris.beams.io/posts/git-commit/

[JF18] Jeff Forcier. 2018. 'Contributing to Open Source Projects.' https://contribution-guide-org.readthedocs.io/

PYTHON MODULE INDEX

S

INDEX

Symbols

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