Adding New Custom Wizards

If you have a team working on a large application or several applications, you might want to standardize the way the team members create projects and files.

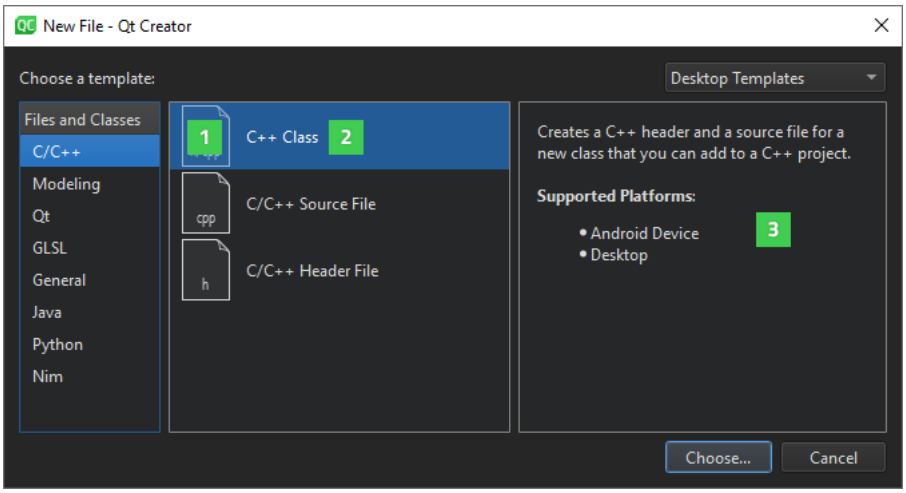
You can create custom wizards in JSON format. They are stored in wizard template directories that contain a JSON configuration file called `wizard.json` and any template files nee
and generators for creating files.

To create a customized wizard, copy a template directory to the shared directory or the local user's settings directory under a new name. Then change the wizard id in the `wizard.j`

You can create a subdirectory for the templates in the settings directory. The standard wizards are organized into subdirectories by type, but you can add your wizard directory to any

To share the wizard with other users, you can create an archive of the wizard directory and instruct the recipients to extract it into one of the directories Qt Creator searches wizards

Qt Creator displays the wizards that it finds in the **New Project** and **New File** dialogs. For each wizard, an icon (1), a display name (2), and a description (3) are displayed.



Wizard Types

In a project wizard, you can specify the files needed in a project. You can add wizard pages to allow developers to specify settings for the project.

A file wizard is similar, but does not contain any project file.

Locating Wizards

Qt Creator searches the following locations for wizards:

- Shared directory:
  - On Windows: `share\qtcreator\templates\wizards`
  - On Linux: `share/qtcreator/templates/wizards`
  - On macOS: `Qt Creator.app/Contents/Resources/templates/wizards`
- Local user's settings directory:
  - On Windows: `%APPDATA%\QtProject\qtcreator\templates\wizards`
  - On Linux and macOS: `$HOME/.config/QtProject/qtcreator/templates/wizards`

Tips for Wizard Development

Assign keyboard shortcuts to some helper actions and turn on verbose output.

Mapping Actions to Keyboard Shortcuts

Qt Creator has some actions that can improve the wizard development process. These are by default not bound to any keyboard shortcuts and can thus not be triggered. To enable th

The following actions can help with wizard development:

Action Id	Description
Action Id	Description
Inspect	Triggering this action opens a window that lists all the defined fields and variables in the wizard at the time the action was triggered. Each activation of this action

Verbose Output

For wizard development, we recommend that you start Qt Creator with the `-customwizard-verbose` argument to receive confirmation that Qt Creator was able to find and pars might run into while editing wizards.

In verbose mode, each correctly set up wizard produces output along the following lines:

```
Checking "/home/jsmith/.config/QtProject/qtcreator/templates/wizards/mywizard"
for wizard.json.
* Configuration found and parsed.
```

The output includes the name of the directory that was checked for a `wizard.json` file. If the file is not found, the message is not displayed.

If the file contains errors, such as an invalid icon path, the following types of messages are displayed:

```
Checking "/home/jsmith/.config/QtProject/qtcreator/templates/wizards/mywizard"
for wizard.json.
* Configuration found and parsed.
* Failed to create: Icon file
"/home/jsmith/.config/QtProject/qtcreator/templates/wizards/mywizard/../..
/global/genericfilewizard.png" not found.
```

See Using Command Line Options for more information about command line arguments.

Integrating Wizards into Builds

To integrate the wizard into Qt Creator and to deliver it as part of the Qt Creator build, place the wizard files in the Qt Creator sources. Then select **Build** > **Run CMake** or **Run qmake**, Creator source directory into the Qt Creator build directory as part of the next Qt Creator build.

If you do not run CMake or qmake, your new wizard will not show up because it does not exist in the build directory you run your newly built Qt Creator from. It never got copied there

Basically, CMake and qmake generate a fixed list of files to copy from the source directory to the subdirectory of the build directory that is checked for wizards at runtime. Therefore,

Using Variables in Wizards

You can use variables (`%\{<variableName>\}`) in strings in the JSON configuration file and in template source files. A set of variables is predefined by the wizards and their pages section in the `wizard.json` file.

There is a special variable `%\{JS:<JavaScript expression>\}` which evaluates the given JavaScript expression and converts the resulting JavaScript value to a string. In the J JavaScript object has the type that the value of the variable has, which can be a string, list, dictionary or boolean.

In places where a boolean value is expected and a string is given, an empty string as well as the string `"false"` is treated as `false` and anything else as `true`.

Localizing Wizards

If a setting name starts with the `tr` prefix, the value is visible to users and should be translated. If the new wizard is included in the Qt Creator sources, the translatable strings appe .json file using the following syntax:

```
"trDisplayName": { "C": "default", "en_US": "english", "de_DE": "deutsch" }
```

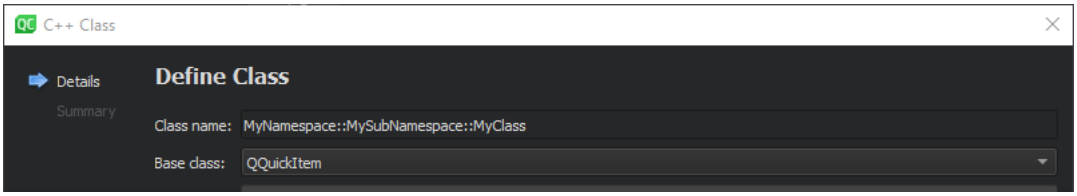
For example:

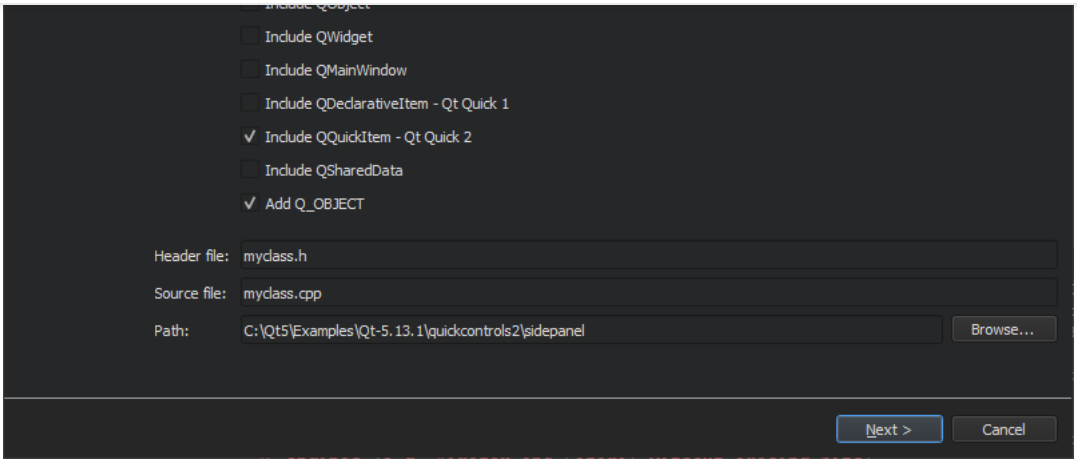
```
"trDisplayName": { "C": "Project Location", "en_US": "Project Location", "de_DE": "Projekt Verzeichnis" }
```

Creating Wizards

Qt Creator contains wizards for adding classes, files, and projects. You can use them as basis for adding your own wizards. We use the C++ wizard to explain the process and the sect

In this example, we create the wizard directory in the shared directory and integrate it in the Qt Creator build system, so that it can deployed along with the Qt Creator binaries as par





For more information about the pages and widgets that you can add and their supported properties, see Available Pages and Available Widgets.

To create a JSON-based C++ class wizard:

1. Start Qt Creator with the `-customwizard-verbose` argument to receive feedback during wizard development. For more information, see Verbose Output.
2. Set keyboard shortcuts for the **Inspect** and **Factory.Reset** actions, as described in Tips for Wizard Development.
3. Make a copy of `share/qtcreator/templates/wizards/classes/cpp` and rename it. For example, `share/qtcreator/templates/wizards/classes/mycp`
4. Use the **Factory.Reset** action to make the wizard appear in **File** > **New File** without restarting Qt Creator.
5. Open the wizard configuration file, `wizard.json` for editing:
   - The following settings determine the type of the wizard and its place in the **New File** dialog:

```
"version": 1,
"supportedProjectTypes": [ ],
"id": "A.Class",
"category": "O.C++",
```

     - `version` is the version of the file contents. Do not modify this value.
     - `supportedProjectTypes` is an optional setting that can be used to filter wizards when adding a new build target to an existing project. For example, only wiza

       Possible values are the build systems supported by Qt Creator or `UNKNOWN_PROJECT` if the build system is not specified: `AutotoolsProjectManager.Auto` `PythonProject`, `Qbs.QbsProject`, `Qt4ProjectManager.Qt4Project` (qmake project), `QmlProjectManager.QmlProject`
     - `id` is the unique identifier for your wizard. Wizards are sorted by the ID in alphabetic order within the `category`. You can use a leading letter to specify the positio

       This information is available in the wizard as `%\{id\}`.
     - `category` is the category in which to place the wizard in the list. You can use a leading letter to specify the position of the category in the list in the **New File** dialo

       This information is available in the wizard as `%\{category\}`.
   - The following settings specify the icon and text that appear in the **New File** dialog:

```
"trDescription": "Creates a C++ header and a source file for a new class that you can add to a C++ project.",
"trDisplayName": "C++ Class",
"trDisplayCategory": "C++",
"iconText": "h/cpp",
"enabled": "%{JS: value('Plugins').indexOf('CppEditor') >= 0}",
```

     - `trDescription` appears in the right-most panel when `trDisplayCategory` is selected.

       This information is available in the wizard as `%\{trDescription\}`.
     - `trDisplayName` appears in the middle panel when `trDisplayCategory` is selected.

       This information is available in the wizard as `%\{trDisplayName\}`.
     - `trDisplayCategory` appears in the **New File** dialog, under **Files and Classes**.

       This information is available in the wizard as `%\{trDisplayCategory\}`.
     - `icon` appears next to the `trDisplayName` in the middle panel when `trDisplayCategory` is selected. We recommend that you specify the path relative to th
     - `iconText` determines the text overlay for the default file icon.
     - `iconKind` determines whether the icon is themed.
     - `image` specifies a path to an image (for example a screenshot) that appears below the `trDescription`.
     - `featuresRequired` specifies the Qt Creator features that the wizard depends on. If a required feature is missing, the wizard is hidden. For example, if no kit has

       Use `enabled` if you need to express more complex logic to decide whether or not your wizard will be available.

       This information is available in the wizard as `%\{RequiredFeatures\}`.

This information is available in the wizard as `%\{PreferredFeatures\}`.

- `platformIndependent` is set to `true` if the wizard is supported by all target platforms. By default, it is set to `false`.
- `enabled` is evaluated to determine whether a wizard is listed in **File** > **New Project** or **New File**, after `featuresRequired` has been checked.

  The default value is `true`.

- The `options` section contains an array of objects with *key* and *value* attributes. You can define your own variables to use in the configuration and template source files, i

```
"options":
[
    { "key": "TargetPath", "value": "%{Path}" },
    { "key": "HdrPath", "value": "%{Path}/%{HdrFileName}" },
    { "key": "SrcPath", "value": "%{Path}/%{SrcFileName}" },
    { "key": "CN", "value": "%{JS: Cpp.className(value('Class'))}" },
    { "key": "Base", "value": "%{JS: value('BaseCB') === '' ? value('BaseEdit') : value('BaseCB')}" },
    { "key": "isQObject", "value": "%{JS: (value('Base') === 'QObject' || value('Base') === 'QWidget' || value('Base') === 'QMainWindo
    { "key": "GUARD", "value": "%{JS: Cpp.classToHeaderGuard(value('Class'), Util.suffix(value('HdrFileName'))}" },
    { "key": "SharedDataInit", "value": "%{JS: value('IncludeQSharedData') ? 'data(new %{CN}Data)' : '' }" }
],
```

This section is optional. For more examples of variables, see the `wizard.json` files for other wizards.

- The `pages` section specifies the wizard pages. The pages used depend on the wizard type. You can add standard pages to wizards or create new pages using the availab

```
"pages":
[
    {
        "trDisplayName": "Define Class",
        "trShortTitle": "Details",
        "typeId": "Fields",
        "data" :
        [
            {
                "name": "Class",
                "trDisplayName": "Class name:",
                "mandatory": true,
                "type": "LineEdit",
                "data": {
                    "trPlaceholder": "Fully qualified name, including namespaces",
                    "validator": "(?:(?:[a-zA-Z_][a-zA-Z_0-9]*::)*[a-zA-Z_][a-zA-Z_0-9]*|)",
                    "completion": "namespaces"
                    }
    },
            ...
]
```

  - `typeId` specifies the page to use: `Fields`, `File`, `Form`, `Kits`, `Project`, `VcsConfiguration`, `VcsCommand` or `Summary`.

    Full page ID, as used in the code, consists of the `typeId` prefixed with `"PE.Wizard.Page."`. For more information, about the pages, see Available Pages.

  - `trDisplayName` specifies the title of the page. By default, the page title is used.
  - `trShortTitle` specifies the title used in the sidebar of the Wizard. By default, the page title is used.
  - `trSubTitle` specifies the subtitle of the page. By default, the page title is used.
  - `index` is an integer value that specifies the page ID. It is automatically assigned if you do not set it.
  - `enabled` is set to `true` to show the page and to `false` to hide it.
  - `data` specifies the wizard pages. In the C++ wizard, it specifies a `Fields` page and a `Summary` page. The `Fields` page contains the `CheckBox`, `ComboBox`, `Li`

- The `generators` section specifies the files to be added to the project:

```
"generators":
[
    {
        "typeId": "File",
        "data":
        [
            {
                "source": "file.h",
                "target": "%{HdrPath}",
                "openInEditor": true
                "options": [
                    { "key": "Cpp:License:FileName", "value": "%{HdrFileName}" },
                    { "key": "Cpp:License:ClassName", "value": "%{CN}" }
                ]
            },
            {
```

```
                "options": [
                    { "key": "Cpp:License:FileName", "value": "%{SrcFileName}" },
                    { "key": "Cpp:License:ClassName", "value": "%{CN}" }
                ]
            }
 ]
```

- `typeId` specifies the type of the generator. Currently, only `File` or `Scanner` is supported.
- `data` allows to configure the generator further.

Values Available to the Wizard

In addition to properties taken from the `wizard.json` file itself (see Creating Wizards), Qt Creator makes some information available to all JSON based wizards:

- `WizardDir` is the absolute path to the `wizard.json` file.
- `Features` lists all features available via any of the kits configured in Qt Creator.
- `Plugins` contains a list of all plugins running in the current instance of Qt Creator.
- `Platform` contains the platform selected in the **File** > **New Project** or **New File** dialog. This value may be empty.

The following information is only available when the wizard was triggered via the context menu of a node in the **Projects** view:

- `InitialPath` with the path to the selected node.
- `ProjectExplorer.Profile.Ids` contains a list of Kits configured for the project of the selected node.

Available Pages

You can add predefined pages to wizards by specifying them in the `pages` section of a wizard.json file.

Field Page

A Field page has the `typeId` value `Field` and contains widgets. For more information about widget definitions, see Available Widgets.

```
"pages":
[
    {
        "trDisplayName": "Define Class",
        "trShortTitle": "Details",
        "typeId": "Fields",
        "data" :
        [
            {
                "name": "Class",
                "trDisplayName": "Class name:",
                "mandatory": true,
                "type": "LineEdit",
                "data": {
                    "trPlaceholder": "Fully qualified name, including namespaces",
                    "validator": "(?:(?:[a-zA-Z_][a-zA-Z_0-9]*::)*[a-zA-Z_][a-zA-Z_0-9]*|)",
                    "completion": "namespaces"
                    }
            },
            ...
],
```

File Page

A File page has the `typeId` value `File`. You can leave out the `data` key or assign an empty object to it.

```
{
    "trDisplayName": "Location",
    "trShortTitle": "Location",
    "typeId": "File"
},
```

The page evaluates `InitialFileName` and `InitialPath` from the wizard to set the initial path and filename. The page sets `TargetPath` to the full path of the file to be create

Form Page

A Form page has the `typeId` value `Form`. You can leave out the `data` key or assign an empty object to it.

```
    "typeId": "Form"
},
```

The page sets `FormContents` to an array of strings with the form contents.

Kits

A Kits page has the `typeId` value `Kits`. The `data` section of a Kits page contains an object with the following settings:

- `projectFilePath` with the path to the project file.
- `requiredFeatures` with a list of strings or objects that describe the features that a kit must provide to be listed on the page.

  When a string is found, this feature must be set. When using an object instead, the following settings are checked:

  - `feature`, which must be a string (that will have all `%\{<VariableName\}` expanded).
  - `condition`, which must evaluate to `true` or `false` and can be used to discount the feature from the list.
- `preferredFeatures` with a list in the same format as requiredFeatures. Any kit matching all features listed in `preferredFeatures` (in addition to `requiredFeature`

```
{
    "trDisplayName": "Kit Selection",
    "trShortTitle": "Kits",
    "typeId": "Kits",
    "enabled": "%{IsTopLevelProject}",
    "data": { "projectFilePath": "%{ProFileName}" }
},
```

The page evaluates `%\{Platform\}` to set the platform selected in **File** > **New Project** or **New File**.

Project

A Project page has the `typeId` value `Project`. It contains no data or an object with the `trDescription` property which will be shown on the generated page. `trDescription`
`wizard.json` file.

```
{
    "trDisplayName": "Project Location",
    "trShortTitle": "Location",
    "typeId": "Project",
    "data": { "trDescription": "A description of the wizard" }
},
```

The page evaluates `InitialPath` to set the initial project path. The page sets `ProjectDirectory` and `TargetPath` to the project directory.

Summary

A Summary page has the `typeId` value `Summary`. It contains no data or an empty object.

```
{
    "trDisplayName": "Project Management",
    "trShortTitle": "Summary",
    "typeId": "Summary"
}
```

The page sets `IsSubproject` to an empty string if this is a toplevel project and to `yes` otherwise. It sets `VersionControl` to the ID of the version control system in use.

VcsCommand

The VcsCommand page runs a set of version control operations and displays the results.

The `data` section of this page takes an object with the following keys:

- `vcsId` with the id of the version control system to be used.
- `trRunMessage` with the message to be shown while the version control is running.
- `extraArguments` with a string or a list of strings defining extra arguments passed to the version control checkout command.
- `repository` with the URL to check out from the version control system.
- `baseDirectory` with the directory to run the checkout operation in.
- `checkoutName` with the subdirectory that will be created to hold the checked out data.
- `extraJobs` with a list of objects defining additional commands to run after the initial checkout. This can be used to customize the repository further by for example adding ad

- `directory` with the working directory of the command to be run. This defaults to the value of `baseDirectory`.
- `command` with the command to be run.
- `arguments` with the arguments to pass to `command`.
- `timeOutFactor` can be used to provide for longer than default timeouts for long-running commands.
- `enabled` which will be evaluated to decide whether or not to actually execute this job.

VcsConfiguration

The VcsConfiguration page asks the user to configure a version control system and only enables the **Next** button once the configuration is successful.

The `data` section of this page takes an object with the `vcsId` key. This setting defines the version control system that will be configured.

Available Widgets

You can add the following widgets on a Field page:

- Check Box
- Combo Box
- Label
- Line Edit
- Path Chooser
- Spacer
- Text Edit

> **Note:** Only the the settings documented in the following sections are supported in wizards.

Specify the following settings for each widget:

- `name` specifies the widget name. This name is used as the variable name to access the value again.
- `trDisplayName` specifies the label text visible in the UI (if `span` is not `true`).
- `type` specifies the type of the widget: `CheckBox`, `ComboBox`, `Label`, `LineEdit`, `PathChooser`, `Spacer`, and `TextEdit`.
- `trToolTip` specifies a tool tip to show when hovering the field with the mouse.
- `isComplete` is evaluated for all fields to decide whether the **Next** button of the wizard is available or not. All fields must have their `isComplete` evaluate to `true` for this to
- `trIncompleteMessage` is shown when the field's `isComplete` was evaluated to `false`.
- `persistenceKey` makes the user choice persistent. The value is taken to be a settings key. If the user changes the default value of the widget, the user-provided value is sto
- `visible` is set to `true` if the widget is visible, otherwise it is set to `false`. By default, it is set to `true`.
- `enabled` is set to `true` if the widget is enabled, otherwise it is set to `false`. By default, it is set to `true`.
- `mandatory` is set to `true` if this widget must have a value for the **Next** button to become enabled. By default, it is set to `true`.
- `span` is set to hide the label and to span the form. By default, it is set to `false`. For more information, see Using Variables in Wizards.
- `data` specifies additional settings for the particular widget type, as described in the following sections.

Check Box

```
{
    "name": "IncludeQObject",
    "trDisplayName": "Include QObject",
    "type": "CheckBox",
    "data":
    {
        "checkedValue": "QObject",
        "uncheckedValue": "",
        "checked": "%{JS: value('BaseCB') === 'QObject' ? 'true' : 'false'}"
    }
},
```

- `checkedValue` specifies the value to set when the check box is enabled. By default, set to `true`.
- `uncheckedValue` specifies the value to set when the check box is disabled. By default, set to `false`.
- `checked` is set to `true` if the check box is enabled, otherwise `false`.

List

> **Note:** The Combo Box and Icon List types are both variations of the List type, and therefore they can have the same properties.

```
    "type": "ComboBox",
    "data":
    {
        "items": [ { "trKey": "<Custom>", "value": "" },
                   "QObject", "QWidget", "QMainWindow", "QDeclarativeItem", "QQuickItem" ]
    }
},
```

or

```
{
    "name": "ChosenBuildSystem",
    "trDisplayName": "Choose your build system:",
    "type": "IconList",
    "data":
    {
        "items": [
            { "trKey": "CMake", "value": "cmake", "icon": "cmake_icon.png", "trToolTip": "Building with CMake." },
            { "trKey": "Qbs", "value": "qbs", "icon": "qbs_icon.png", "trToolTip": "Building with Qbs." },
            { "trKey": "QMake", "value": "qmake", "icon": "qmake_icon.png", "trToolTip": "Building with QMake." }
        ]
    }
},
```

- `items` specifies a list of items to put into the list type. The list can contain both JSON objects and plain strings. For JSON objects, define `trKey` and `value` pairs, where the `tr` specify an icon for the list item and `trToolTip` to specify a tooltip for it.
- `index` specifies the index to select when the list type is enabled. By default, it is set to 0.
- `disabledIndex` specifies the index to show if the list type is disabled.

Label

```
{
    "name": "LabelQQC_2_0",
    "type": "Label",
    "span": true,
    "visible": "%{JS: value('CS') === 'QQC_2_0'}",
    "data":
    {
        "wordWrap": true,
        "trText": "Creates a deployable Qt Quick 2 application using Qt Quick Controls.",
    }
},
```

- `wordWrap` is set to `true` to enable word wrap. By default, it is set to `false`.
- `trText` contains the label text to display.

Line Edit

```
{
    "name": "Class",
    "trDisplayName": "Class name:",
    "mandatory": true,
    "type": "LineEdit",
    "data": {
        "trPlaceholder": "Fully qualified name, including namespaces",
        "validator": "(?:(?:[a-zA-Z_][a-zA-Z_0-9]*::)*[a-zA-Z_][a-zA-Z_0-9]*|)",
        "completion": "namespaces"
    }
},
{
    "name": "BaseEdit",
    "type": "LineEdit",
    "enabled": "%{JS: value('BaseCB') === '' ? 'true' : 'false'}",
    "mandatory": false,
    "data":
    {
        "trText": "%{BaseCB}",
        "trDisabledText": "%{BaseCB}",
        "completion": "classes"
    }
},
```

- `trDisabledText` specifies the text to display in a disabled field.
- `completion` lists existing `namespaces` for the class name line edit and existing `classes` for the base class line edit. This value replaces the history completer that is usual
- `trPlaceholder` specifies the placeholder text.
- `validator` specifies a QRegularExpression to validate the line edit against.
- `fixup` specifies a variable that is used to fix up the string. For example, to turn the first character in the line edit to upper case.
- `isPassword` is a boolean value that specifies that the line edit contains a password, which will be masked.
- `historyId` is a key that specifies the name for a list of items for the history completer. This value and `completion` are mutually exclusive, so do not set both of them at the
- `restoreLastHistoryItem` is a boolean that specifies that the last history item is automatically set as the default text in the line edit. This key can only be set to true if `hi`

Path Chooser

```
{
    "name": "Path",
    "type": "PathChooser",
    "trDisplayName": "Path:",
    "mandatory": true,
    "data":
    {
        "kind": "existingDirectory",
        "basePath": "%{InitialPath}",
        "path": "%{InitialPath}"
    }
},
```

- `path` specifies the selected path.
- `basePath` specifies a base path that lookups are relative to.
- `kind` defines what to look for: `existingDirectory`, `directory`, `file`, `saveFile`, `existingCommand`, `command`, or `any`.

Spacer

```
{
    "name": "Sp1",
    "type": "Spacer",
    "data":
    {
        "factor": 2
    }
},
```

The `factor` setting specifies the factor (an integer) to multiply the layout spacing for this spacer.

Text Edit

```
{
    "name": "TextField",
    "type": "TextEdit",
    "data" :
    {
        "trText": "This is some text",
        "richText": true
    }
}
```

- `trText` specifies the text to display.
- `trDisabledText` specifies the text to display when the text edit is disabled.
- `richText` is set to `true` for rich text, otherwise `false`.

Available Generators

Qt Creator supports two different generators for JSON wizards.

File Generator

A `File` generator expects a list of objects in its `data` section. Each object defines one file to be processed and copied into the `%\{TargetPath\}` (or any other location).

Each file object can take the following settings:

- `source` specifies the path and filename of the template file relative to the directory containing the `wizard.json` file.

- `target` specifies the location of the generated file, either absolute or relative to `%{TargetPath}`, which is usually set by one of the wizard pages.
- `openInEditor` opens the file in the appropriate editor if it is set to `true`. This setting defaults to `false`.
- `openAsProject` opens the project file in Qt Creator if it is set to `true`. This setting defaults to `false`.
- `isBinary` treats the file as a binary and prevents replacements from being done in the file if set to `true`. This setting defaults to `false`.
- `condition` generates the file if the condition returns `true`. This setting defaults to `true`. For more information, see Using Variables in Wizards.

Scanner Generator

A `Scanner` generator scans the `%\{TargetPath\}` and produces a list of all files found there.

The `Scanner` generator takes one object in its `data` section with the following settings:

- `binaryPattern` is a regular expression that will be matched against all file names found. Any match will be marked as a binary file and template substitution will be skipped
- `subdirectoryPatterns` is a list of regular expression patterns. Any directory matching one of these patterns will be scanned as well as the top level directory. This setting

‹ Adding Libraries to Projects