Shift - Developer guide

Gilles Grousset

July 21, 2013

Revision history

Author	Date	Comment
Gilles Grousset	07/21/2013	Initial version (draft)

Intended audience

This guide is aimed at developers willing to contribue to the main application.

For plugin development, please refer to the **Plugin authoring guide**.

Contents

1 Getting started **5**

1.1 Required skills . . . 5

1.2 Technical setup and tools . . . 5

1.2.1 Java . . . 5

1.2.2 Maven . . . 5

1.2.3 IDE . . . 5

1.3 Building from sources . . . 6

1.3.1 Running the application . . . 6

1.3.2 Packaging the application . . . 6

1.4 Conventions . . . 6

1.4.1 Coding conventions . . . 6

1.4.2 Naming conventions . . . 6

1.5 Overview & concepts . . . 7

1.5.1 Model and managers . . . 7

1.5.2 GUI . . . 7

1.6 Application lifecycle . . . 7

1.6.1 Startup sequence . . . 7

1.6.2 Shutdown sequence . . . 8

1.6.3 ApplicationContext . . . 8

2 The workspace **9**

2.1 HTTP Proxy . . . 9

2.2 Artifacts . . . 10

2.3 Observing the workspace . . . 10

2.3.1 Artifact case . . . 11

2.3.2 Workspace case . . . 11

2.3.3 Knowing what changed on the observable . . . 11

3 Managers **13**

3.1 Preferences . . . 13

3.1.1 Transactions . . . 13

3.1.2 Setting initial values . . . 13

3.2 States . . . 14

3.2.1 Saving state . . . 14

3.2.2 Restoring state . . . 14

3.2.3 Instances identification . . . 15

3.3 Tasks . . . 15

3.3.1 Shared TaskManager . . . 15

3.3.2 Creating a TaskManager . . . 15

3.3.3 Creating a new task . . . 15

3.3.4 Listening to a TaskManager . . . 16

3.4 Plugins . . . 17

3.4.1 Plugins loading . . . 17

3.4.2 Groovy DSL (declaration) . . . 18

3.4.3 Editors . . . 20

3.4.4 Project wizards . . . 21

3.4.5 Previews . . . 21

3.5 Themes . . . 21

4 User interface **23**

4.1 AbstractController and AbstractDialogController . . . 23

4.2 Main window . . . 23

4.2.1 Menu . . . 23

4.2.2 Project navigator . . . 23

4.2.3 Editors pane . . . 23

4.2.4 Status bar . . . 23

4.3 Validation . . . 24

4.3.1 CompoundValidator . . . 24

4.3.2 Validation result . . . 24

4.4 Custom controls . . . 24

Chapter 1

Getting started

1.1 Required skills

The application is developped using Java and JavaFX, Maven is used for building.

Thereby a good knowledge of Java and Object Oriented Programming is required, as well as Asynchronous Programmation.

1.2 Technical setup and tools

1.2.1 Java

The required JDK version is 1.7 (a recent official Oracle version with JavaFX 2.2 included).

However, due to some limitations with Java 1.7, Java 1.8 will be used as soon as it will be available.

The current limitations on Java 1.7 are:

- High-density screens (such as MacBook Pro Retina) are not supported: this results in blurry windows on those screens.
- JavaFX `WebView` does not allow the change of the user agent: this could be a limitation for previewing HTML pages in the future.

1.2.2 Maven

Maven 3 and the javafx-maven-plugin are used to run and package the application.

1.2.3 IDE

Any IDE can be used for development, however Netbeans is recommended as it has the best JavaFX support at the moment.

1.3 Building from sources

1.3.1 Running the application

In order to run the application use:

```
mvn jfx:run
```

1.3.2 Packaging the application

Packaging means : building the application and a native installer for it. The format depends on the operating system and additional tools may be installed prior to creating the package.

Read Building a Native Installer for more information.

The command for packaging is:

```
mvn clean jfx:native
```

1.4 Conventions

1.4.1 Coding conventions

Generic Oracle Java conventions are used for the project: Code Conventions for the Java TM Programming Language

1.4.2 Naming conventions

1.4.2.1 FXML

FXML files are located into `fxml` resource package and named using the underscore convention.

FXML file name must be the same as its controller class.

For example, the FXML file name for `HTMLPreviewController`is `html_preview.fxml`

1.4.2.2 CSS

CSS class names must use dashes and ids use the CamelCase.

Example:

```
#projectNavigator {
    ...
}

.text-input .invalid {
    ...
}
```

1.5 Overview & concepts

1.5.1 Model and managers

The application is built around code editor / IDE core concepts:

- **Workspace**: the place where every artifact lives. An artifact can be any piece of data that can be edited or created from the application : a project is an artifact, a folder or a source file is also an artifact.
- **Preferences**: a centralized way to manage application settings / preferences.
- **States**: the ability to save and restore and object state upon application restart.
- **Tasks**: task queue to mange asynchronous operations.
- **Plugins**: to make the application extensible.
- **Themes**: look and feel management.

1.5.2 GUI

The user interface (GUI) is built upon (in a same main window):

- **Menu**: application main menu.
- **Project navigator**: tree navigation into the workspace and interactions with it.
- **Editors pane**: holds tabs for documents opened into editors.
- **Status bar**: gives information about task being performed, current document…

Child windows can be opened from the main windows, such as : preview windows, wizard windows, new file / folder window…

1.6 Application lifecycle

The application is started by the `com.backelite.shift.MainApp` class (`start(…)` method). When it is shuting down, the method `stop(…)` of the same class is called.

1.6.1 Startup sequence

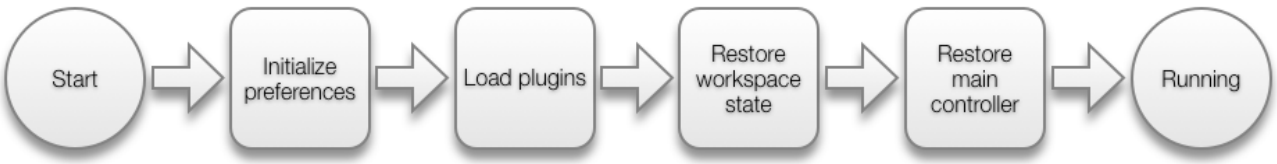

Figure 1.1: Startup sequence diagram

When starting, the application first initializes preferences: it checks if all required settings are set (and set them with a default value if they aren't). See Preferences section for more information.

After that it loads plugins. See Plugins section for more information.

Then it restores the workspace state. See States section for more informations.

And finally, it brings up the main window interface and restore its state as well.

1.6.2 Shutdown sequence

Figure 1.2: Shutdown sequence diagram

At shutdown, the workspace state is saved, the main controller state is saved and the `ApplicationContext` is destroyed.

1.6.3 ApplicationContext

`com.backelite.shift.ApplicationContext` is the main entry point to access application centralized components.

It acts as a holder for storing singleton instances of application components, such as the workspace, the plugin manager, …

For example, it is possible to access the plugin manager like this from anywhere in the application:

```
ApplicationContext.getPluginManager();
```

Chapter 2

The workspace

Classes related to the workspace are located in `com.backelite.shift.workspace`.

The workspace is represented by the `Workspace` interface. The default workspace implementation is `LocalWorkspace`.

This implementation manages artifacts from the local file system.

`Workspace` extends the `PersistableState` interface in order to be able to save / restore the current workspace on application exit / restart.

The application only holds one workspace (singleton). It can be accessed using `com.backelite.shift.ApplicationC` static method:

```
ApplicationContext.getWorkspace();
```

2.1 HTTP Proxy

When required (usually by previewers), the `HTTPWorkspaceProxyServer` (singleton) can be started in order to serve workspace artifacts on the HTTP protocol.

The HTTP proxy can be accessed using `com.backelite.shift.ApplicationContext` static method:

```
ApplicationContext.getHTTPWorkspaceProxyServer();
```

It can be started like this (ignored if the server is already started):

```
ApplicationContext.getHTTPWorkspaceProxyServer().start();
```

It can be stopped with (automatically stopped at application shutdown):

```
ApplicationContext.getHTTPWorkspaceProxyServer().stop();
```

The port of the server is determined at runtime (on start) depending on port number availability (see `com.backelite.shift.util.NetworkUtil.findAvailablePort(…)` for more information).

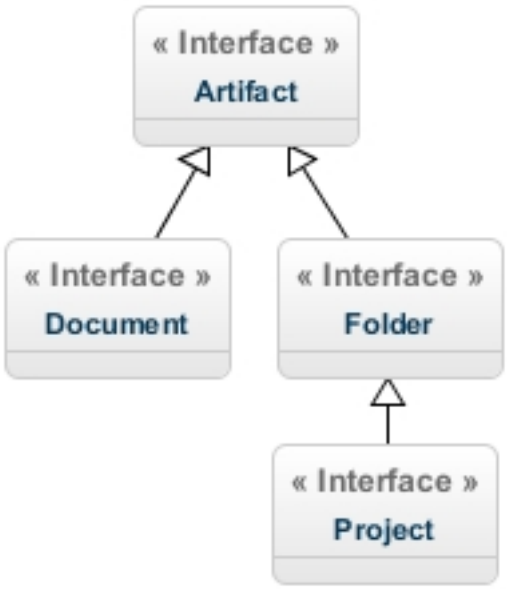

Figure 2.1: Artifact inheritance diagram

2.2 Artifacts

Classes related to workspace artifacts are located in `com.backelite.shift.workspace.artifact`.

`Artifact` interface is the most basic representation of a workspace item. An artifact can be (at the moment) a `Project`, a `Folder` or a `Document` (interfaces).

`Project` interface inherits the `Folder` interface: a project is a folder with extra attributes and features.

Concrete implementations of those artifacts are `FileSystemProject`, `FileSystemFolder` and `FileSystemDocument`.

> *Note: a file is referred as a* `Document` *artifact to avoid the confusion with* `java.io.File`*, in the GUI layer however it is designated as file to follow editors / IDE conventions.*

2.3 Observing the workspace

In order to make the GUI reflect workspace updates (new folder created, document modified…), every `Workspace` or `Artifact` implementation is observable which mean that they inherit `fr.grousset.util.WeakObserv` object.

`WeakObservable` is an enhanced implementation of the original `java.util.Observable` but that does not require to unregister observers as they are kept as weak references.

For instance, when a GUI controller needs to be notified of an artifact update it only needs to implements the `java.util.Observer` interface and register itself as observer on the artifact:

```
document.addObserver(this);
```

When implementing `java.util.Observer` interface, the following method needs to be defined in order to receive update notifications:

```
public void update(Observable o, Object arg) {
  // Handle change here
}
```

2.3.1 Artifact case

In the artifact case, the `Observable` object is the artifact that was updated (the source) and the arg `Object` (not always defined) is the original source.

For example, the original source is set in case of a `Document` artifact : when it changes, it notifies its observers, but also triggers a notification from its parent folder.

When a `Folder` receives a notification, it forwards the notification with the original source to its observers (at least its parent folder, a `Folder` always observes its children).

Here are practical cases :

Observed artifact	Modified artifact	Observable parameter received	arg Object parameter received
Document	same Document	Document	null
Folder / Project	same Folder / Project	Folder / Project	null
Folder / Project	child or grand-child Document	Folder / Project	child or grand-child Document
Folder / Project	child or grand-child Folder	Folder / Project	child or grand-child Folder

2.3.2 Workspace case

In the workspace case it is simplier: the `Observable` object is always the `Workspace` and the arg `Object` is always the `Project` that was modified.

2.3.3 Knowing what changed on the observable

When `java.util.Observer.update(…)` method is called it is possible to know which artifact changed, but not directly what changed.

To do so, here are some snippet to detect changes:

- `Document` was modified

```
public void update(Observable o, Object arg) {

        // A document was updated
        if (o instanceof Document) {
            Document document = (Document)o;
            if (document.isModified()) {
                // Handle change here
            }
        }
}
```

- `Document` or `Folder` was deleted

```
public void update(Observable o, Object arg) {

        // An artifact was updated
        if (o instanceof Artifact) {
            Artifact artifact = (Artifact)o;
            if (artifact.isDeleted()) {
                // Handle change here
            }
        }
}
```

- `Project` was closed

```
public void update(Observable o, Object arg) {

        // A project was updated
        if (o instanceof Workspace) {
            Project project = (Project)arg;
            if (!ApplicationContext.getWorkspace().getProjects().contains(project) {
                // Handle change here
            }
        }
}
```

Chapter 3

Managers

This part deals with services / managers common to the whole application.

3.1 Preferences

Preferences are used to manage application settings across the application.

Classes related to preferences are located inside `com.backelite.shift.preferences`.

The heart of the preferences system is the `PreferencesManager`interface: it defines methods to store and retrieve preferences (`setValue(…)`, `getValue(…)`).

The default (and unique) implementation of `PreferencesManager`, at the moment, is `LocalPreferencesManager`: this implementation is designed to store preferences as a JSON file *preferences.json* in a local directory.

3.1.1 Transactions

`PreferencesManager` is transactionnal, which means that the `commit()`method must be called to persist modifications. If modifications must be reverted to the last commit, the method `rollback()` can be used.

A singleton `PreferencesManager`instance is used for the whole application and can be accessed from the `ApplicationContext`:

```
PreferencesManager preferencesManager = ApplicationContext.getPreferencesManager();
// Set an initial value
preferencesManager.setInitialValue(”key”, ”value”);
// Set a value
preferencesManager.setValue(”key2”, ”value2”);
preferencesManager.commit();

// Read a value
String value = (String)preferencesManager.getValue(”key3”);
```

3.1.2 Setting initial values

When the application (or a plugin) starts form the first time: it needs to set initial values into preferences, but must no erase existing values if they are already set.

To do so, the `PreferencesManager` exposes the `setInitialValue(…)`(to set a single value) and `setInitialValues(…)` (to set several values at the same time).

When the application starts it checks and initializes initial values with thoses methods. The code is located inside `com.backelite.shift.MainApp.initializePreferences()` and it is called from `com.backelite.shift.MainApp.start(…)`.

See Application lifecycle for more informations about the application startup sequence.

3.2 States

States are used to save and restore application components states between application restarts.

Classes relating to states management are located inside `com.backelite.shift.state`.

Any object of the application can be persisted, it just needs to implement the `PersistableState` interface.

`PersistableState` objects can then be handled by a `StateManager` (interface). The `StateManager`is in charge of serializing and deserializing states to a storage device.

The default (and only one at the moment) concrete implementation of `StateManager` is the `LocalStateManager`: an implementation that stores states as JSON files into a directory.

The default instance of the `StateManager` can be accessed from the `ApplicaitonContext`:

```
ApplicationContext.getStateManager();
```

3.2.1 Saving state

Saving a state consists of populating a `Map<String, Object>` object with all the values required to restore the current object state:

```
public void saveState(Map<String, Object> state) throws StateException {

        state.put(”value1”, this.value1);
        state.put(”value2”, this.value2);
        ...
}
```

3.2.2 Restoring state

Restoring a state consists of reading a `Map<String, Object>` object in order to restore the current object state:

```
public void restoreState(Map<String, Object> state) throws StateException {

    String value1 = (String)state.get(”value1”);
    String value2 = (String)state.get(”value2”);
    ...
}
```

3.2.3 Instances identification

The `PersistableState`interface provides a method named `getInstanceIdentifier()` and is used to identify a given object instance.

If this method returns `null`, a single state is maintianed for all instances of the class. For example, with the `LocalStateManager` saving an object called `MyObject` without providing an identifier will result into a state file name `myobject.json`.

However, if the `getInstanceIdentifier()` returns a value, a state will be maintaned for every instance of the class. In the case of the `LocalStateManager`the state file name for a given instance of `MyObject` will be named `myobjectMYID.json` (where *MYID* is the identifier returned by the instance).

3.3 Tasks

Tasks are used in the application to handle asynchronous processing. They are commonly used to handle time consuming operations without blocking the user interface.

Class relating to tasks and task management are located inside the `com.backelite.shift.task`.

Tasks used are common `javafx.concurrent.Task` objects. There are processed by a `TaskManager`.

The `TaskManager` holds a task queue where new tasks can be added. The sate of a `TaskManager`can be listened by adding a `TaskManagerListener` on it.

The default concrete implementation of `TaskManager`is `LocalTaskManager`.

3.3.1 Shared TaskManager

A shared `TaskManager` can be accessed using `com.backelite.shift.ApplicationContext` static method:

```
ApplicationContext.getTaskManager();
```

This `TaskManager`must be used as much as possible to run tasks: as the central `TaskManager`it is linked to the Status bar GUI element and state (name and progress) of the current running task is displayed on it.

3.3.2 Creating a TaskManager

If for some reasons, a task needs to run outside the sahre `TaskManager` (to run a background download without preventing the user to work), it is possible to create a `LocalTaskManager` like this (usually in a controller class):

```
TaskManager taskManager = new LocalTaskManager();
```

3.3.3 Creating a new task

Adding a task to a `TaskManager` can be done like this:

```
ApplicationContext.getTaskManager().addTask(new Task() {
    @Override
    protected Object call() throws Exception {

        // Set title
        updateTitle("task title");

        // Let's say the process iterates over an item array
        for (int i = 0; i < 10; i++) {

            // Process item here…
            …

            // Update progress
            updateProgress(i + 1, 10);
        }

        // Return anything…
        return true;

    }
});
```

The `updateTitle(…)` method can be called to give the task a name. If the shared `TaskManager`is used, this name will be displayed in the status bar while the task is running.

The `updateProgress(…)` gives the ability to track the task progress. Once again: if the shared `TaskManager`is used, the progress will be displayed as a progress bar in tge status bar while the task is running.

3.3.4 Listening to a TaskManager

A `TaskManager`can be listen to by implementening the `TaskManagerListener`, and registering the listener on the `TaskManager`.

A `TaskManagerListener` gets notified when:

- A task is started
- A task succeeded
- A task failed

Here is an example showing how to create a task and know when it has completed:

First register the current class as listener with:

```
ApplicationContext.getTaskManager().addListener(this);
```

Then, create the task and add it to the manager (and keep it in a variable):

```
Task myTask = new Task() {…};
ApplicationContext.getTaskManager().addTask(myTask);
```

Finally in the `onTaskSucceeded(…)` method, intercept the task:

```
public void onTaskSucceeded(Task task) {
    if (task == myTask) {
        // Do whatever you want here
        ...
    }
}
```

3.4 Plugins

As many code editors / IDE these days, many features of the application are provided by plugins.

At the moment, a plugin can provide the following items:

- **Editors**: editors are pieces of GUI that allow the display and modification of document artifact (source code, image or any other resource file).
- **Wizards**: wizards are pieces of GUI used to control code / files generation. Example : a HTML project wizard provides GUI and code to generate an initial HTML project structure.
- **Previews**: previews are pieces of GUI in charge of rendering a document or project.

Classes related to plugins are located inside the `com.backelite.shift.plugin` package.

Plugins (plugin declarations) are represented by the `Plugin` class. They are held into a `PluginRegistry`.

At the moment, plugin declarations have to be written in Groovy JVM language.

The default implementation of `PluginRegistry` is `LocalPluginRegistry`. The `LocalPluginRegistry` loads plugin definitions from a local directory.

`AbstractPluginRegistry` is the base implementation class. However, to leave the possibility to support other languages than Groovy for plugin declaration in the future, Groovy releated code is encapsulated inside the `AbstractGroovyPluginRegistry`.

The singleton instance of the `PluginRegistry` can be accessed from the `com.backelite.shift.ApplicationCon` this way:

```
ApplicationContext.getPluginRegistry();
```

3.4.1 Plugins loading

Plugins are loaded at application startup as explained in Application lifecycle.

Loading plugins involves:

- Loading the built-in plugin
- Loading external plugins

The built-in plugin is a plugin embedded into the application to provide initial editors, wizards and previews. Its declaration file is `Builtin.groovy`, located at the root of the resources directory.

> *Note: at the moment, the external plugin loading feature is not implemented and the* `PluginManager` *loads the built-in plugin only.*

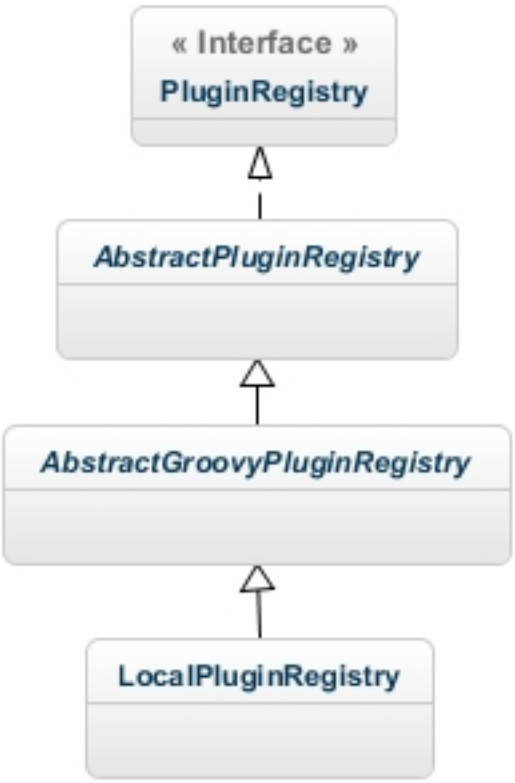

Figure 3.1: PluginRegistry inheritance diagram

3.4.2 Groovy DSL (declaration)

A plugin is declared using a Groovy file, and can define the following:

- Plugin general info: such as identifier, name, version…
- Lifecycle event listeners
- Editor factories
- Preview factories
- Project wizard factories

A Groovy DSL (Domain Specific Language) is used for the declaration. The definition, validation and parsing of this DSL is perfomred by the MetaBuilder Groovy library.

The *MetaBuilder* works with a model declaration schema (like a XSD for an XML file) using a DSL itself. The schema file is located at the root of the resources directory and is named `PluginSchema.groovy`.

Here is the file (this may evolve in time to support new features):

```
import groovytools.builder.MetaBuilder

def metaBuilder = new MetaBuilder(getClass().getClassLoader())

metaBuilder.define {
    plugin(factory: com.backelite.shift.plugin.Plugin) {
        properties {
```

```
        uid(req: true)
        name(req: true)
        description(req: false)
        author(req:false)
        versionCode(req: true)
        versionName(req: true)
        lifecycle(req: false, schema: 'lifecycle')
    }
    collections {
        editorFactories {
            editorFactory(schema: 'editorFactory')
        }
        previewFactories {
            previewFactory(schema: 'previewFactory')
        }
        projectWizardFactories {
            projectWizardFactory(schema: 'projectWizardFactory')
        }
    }
}

editorFactory(factory: com.backelite.shift.plugin.GroovyEditorFactory) {
    properties {
        name(req: true)
        description(req: true)
        supportedExtensions(req: true)
        code(req: true)
    }
}

previewFactory(factory: com.backelite.shift.plugin.GroovyPreviewFactory) {
    properties {
        name(req: true)
        description(req: true)
        supportedExtensions(req: true)
        code(req: true)
    }
}

projectWizardFactory(factory: com.backelite.shift.plugin.GroovyProjectWizardFact
    properties {
        name(req: true)
        description(req: true)
        code(req: true)
        projectGenerator(req: true, schema: 'projectGenerator')
    }
}

projectGenerator(factory: com.backelite.shift.plugin.GroovyProjectGenerator) {
    properties {
```

```
            code(req: true)
        }
    }

    lifecycle(factory: com.backelite.shift.plugin.GroovyPluginLifecycle) {
        properties {
            onLoad(req: false)
        }
    }
}

return metaBuilder
```

3.4.3 Editors

Editors are the corner stone of the application: they provide a UI component to view and edit workspace documents.

To declare a new editor type in a plugin declaration file, a factory closure must be provided. Here is an example to declare a HTML editor (taken from `BuiltinPlugin.groovy`):

```
 editorFactory {
    name = "HTML editor"
    description = "Builtin HTML editor"
    supportedExtensions = ['html']
    code = {document, loader ->
        Node node = (Node) loader.load(getClass().getResourceAsStream("/fxml/code_ed
        CodeEditorController controller = (CodeEditorController) loader.getControlle
        controller.setDocument(document)
        controller.setMode(CodeEditorController.Mode.HTML)
        return node
    }
}
```

An editor factory must define 5 blocks:

1. *name*: the name of the editor type
2. *description*: the description of the editor type
3. *supportedExtensions*: array containing file extensions the editor applies to
4. *code*: the factory code to build the editor given a context `com.backelite.shift.workspace.artifact.Docu` instance and a `javafx.fxml.FXMLLoader`, it must return a `javafx.scene.Node`.

The controller bound to the `javafx.scene.Node` built by the factory must implement the `EditorController` interface.

The factory is called to build an editor when it is requested by the `EditorsPaneContoller.openDocument(…)`.

The choice of the editor to build is made by the `PluginRegistry` in the `newEditor(…)` method: this method looks at the current document extension and finds the first matching editor in the registry.

To make sure, an editor can be found even if the file extension is not supported, a basic text editor is declared into the built-in plugin and uses a wildcard in *supportedExtensions* array:

```
 editorFactory {
    name = "Generic text editor"
    description = "Builtin Text editor"
    supportedExtensions = ['*']
    code = {document, loader ->
        Node node = (Node) loader.load(getClass().getResourceAsStream("/fxml/code_ed
        CodeEditorController controller = (CodeEditorController) loader.getControlle
        controller.setDocument(document)
        return node
    }
}
```

3.4.4 Project wizards

TODO

3.4.5 Previews

TODO

3.5 Themes

TODO

Chapter 4

User interface

TODO

4.1 AbstractController and AbstractDialogController

TODO

4.2 Main window

TODO

4.2.1 Menu

TODO

4.2.2 Project navigator

TODO

4.2.3 Editors pane

TODO

4.2.4 Status bar

TODO

4.3 Validation

Unfortunately, JavaFX does not provide any input validation system at the time. That is why the application uses its own.

Classes related to validation are located in `com.backelite.shift.gui.validation`.

The validation mechanism is pretty basic: it exposes a `Validator` interface with a single `validate(…)` method and a set of implementations to perform common validations : not blank, filename…

Validators are used in Custom controls.

4.3.1 CompoundValidator

The `CompoundValidator` is a special validator that can run several validators at the same time. It can be built like this:

```
Validator validator = new CompoundValidator(new NotBlankValidator(),
new FilenameValidator());
```

4.3.2 Validation result

The `validate(…)` method of `Validator`interface returns a `ValidatorResult` object. This object is composed of a `valid` attribute and a list of error messages (many messages can be returned with the `CompoundValidator`).

Error messages are simple strings formatted as property key, so that it can be localized in a properties file. Example: the `NotBlankValidator` returns the following message: *validator.not.blank*.

4.4 Custom controls

TODO