Saddle technical documentation

Table of Content

1. Required knowledge

To understand this document better it is advised to read through “The Definitive Guide to NetBeans Platform 7” by Heiko Böck, Part 1 “Basics and Concepts” to get a short overview over the NetBeans way of implementing some frequently needed things. This book is generally a good starting point concerning the development of new features that are based on NetBeans functionality and should be read by any new long term developer.

2. Modular structure

Saddle is built on top of the NetBeans platform (currently 7.3). The structure is based on modules for the different functionalities. In general one module should provide one piece of functionality in the UI (like designing a template or workflow) or provide utility classes for several other modules to rely on (like the saddle framework).

Modules may rely on each other and one module may not work if a module dependency is not present. These tight dependencies should only be created if the dependent module really provides essential functionalities without which the main module will not be able to work.

In some cases it might be better to use loose coupling between modules if some functionality should be optional or to avoid circular dependencies between modules.

There is even the possibility to have an implementation dependency on another module to get access to all of the classes in its packages. However these dependencies should be kept as few as possible as they will only work with the one specific version of the dependent plugin and may break during an update. However Saddle uses these dependencies in some places to gain access to NetBeans internal classes. As Saddle does not have an auto update mechanism for the moment the linkage may only break if the NetBeans platform is updated.

3. Lookups and loose coupling

References between modules can also be done by using another module as a dependency or via loose coupling using a structure called Lookups in NetBeans.

These Lookups can best be understood as a Map with classes as keys and multiple implementations of these classes as values. A very common case in NetBeans is that the basic framework asks an instance of a window for an implementation for saving the content of the window to the disk. This happens by querying the Lookup of the window for an object that implements the interface Savable. To provide such an implementation the developer must only put the implementation into the lookup of the created window and thus makes it available to all callers.

This way of loose coupling allows other modules to use functionality in modules that they do not know in detail. The caller just has to define an interface and look for this interface in the corresponding lookups.

Saddle mostly uses this for two tasks: To make functionalities available to the NetBeans framework (like saving, opening files, creating new ones etc.) or to create loose links to our own modules.

In the second case the framework defines interfaces that can be implemented to provide functionality for one specific module like the workflow designer to the interface or other modules without having to make the other modules actually dependent on the workflow designer.

In order for this to work the workflow designer has to register an implementation instance of the interface into the global lookup using the layer.xml file. One example from the workflow designer is the class lu.tudor.santec.saddle.workflowdesigner.project.WorkflowProvider which can also be found in the layer.xml of the workflow provider module.

Any module can now look up instances of the implemented interface WorkflowModuleProvider in the global lookup and will get all implementations registered by other modules. The calling module does not have to know the implementing modules at all. It will just use the implementations by their interface (as defined in the framework) and should also be able to work if there are no implementations in the global Lookup (in this case the best way is to just disable the functionalities).

The framework uses this to generate module specific nodes and actions inside the project tree. As all modules depend on the framework the framework cannot depend on them again as this would cause a circular dependency, so there is a real need for this kind of linking.

4. Important modules in more detail

4.1. Framework

The Saddle framework provides all functionalities that should be available to more than one other module and also provides all basic implementations that are needed to run Saddle. The implementation includes:

- The Saddle project type and all associated actions like the creation and display of Saddle projects. The project template is defined in the file SaddleProject.zip which will be unpacked and modified according to the given parameters once a new project instance has been created.
- The basic data objects and loaders for all types of Saddle files (mule configurations, templates and mappings). These classes also contain automatic upgrade mechanisms before actually loading a file (see ConfigDataObject's constructor). Further automatic upgrades should be implemented in the constructors of the data object classes to make sure the new data object is in an updated state immediately after it has been created.
- Abstract super classes for all TopComponents of Saddle. All editor type windows of Saddle are derived from AbstractSaddleTopComponent and contain AbstractSaddleSubComponents to display the actual content. Adding multiple SubComponents to one TopComponent will result in a tabbed interface, but is not used in Saddle any more. Instead the sub windows like the template tester have been transformed into own TopComponent implementations.

- ModuleProvider interfaces for loose coupling to all UI relevant modules (Message Mapper, Workflow Designer and Template Designer). These interfaces can be used for lookup in the global Lookup as discussed above.

4.2. Message Lib

The message library holds the core of Saddle's parsing functionality, consisting of several transformers and the Field class which is used to contain the parsed representation of a message. Instances of Field can easily be traversed by using the given methods. They have complete JavaDoc and the zipped up JavaDoc is automatically deployed with Saddle.

Another important class in the message library is StructureAttributes which holds constant names and default values for all defined properties inside a template. The naming convention here consists of a prefix and the actual name of the property. The prefix can be one of the following:

- P – For properties affecting the way a message is parsed
- F – For properties affecting the way a certain field inside the message is parsed and validated
- V – For properties affecting the way values are parsed into a field and the validation of the field content

4.3. Workflow Designer

The workflow designer module provides everything necessary to parse, display and modify Mule configuration files.

The workflow designer also initializes the palettes for all mule versions on the first start. The initialized data will be serialized to disk once it has been completed and will be read from there on the next start to have a shorter initialization time. If any error is found while deserializing the whole data will be removed and recreated. To force a rebuild of this data the subfolders of the folder "libraries/palette" representing the mule versions have to be deleted.

Loading a workflow is done by one of two parsers, depending on whether the configuration is based on flows or services. The selection of the correct parser is done automatically and each parser just parses a complete configuration into the currently opened WorkflowTopComponent. The parsers can also be used to write an opened configuration to the disk.

The WorkflowTopComponent (and its WorkflowSubComponent) displays the workflow using NetBeans' own graphic library, the NetBeans Visual Library framework. It allows for layered graphs with a separation of the UI (given by NetBeans) and the logic (given by Saddle) behind the scenes. The core of Saddle's implementation of this framework are the classes WDGraph, BeanWrapperElement and WDEdge representing the graph scene, one node and one edge on the logical level. The graphical representation is just built in memory by the WDGraph using widgets.

To manage the links between the different kinds of elements a set of strategies has been created. Each Strategy handles links going out of one type of element (like transformers, inbound endpoints etc.) to all other possible types of elements. The strategies are used to link the elements together in a live graph instance as well as when everything is written out to the hard disk.

The Workflow Designer module is kept Mule version independent where possible by rather using java reflection calls instead of the specific objects for a given Mule version. In most cases a differentiation between Mule versions is not necessary, as the method names rarely change. For the cases where a differentiation is necessary some objects implement the interface MuleVersionAware that can be used to detect the current mule version for which the object has been created and do the necessary method calls according to the currently used Mule version.

The Workflow Designer uses the aforementioned Lookup objects to make calls to the Template Designer and Message Mapper to open templates and mappings directly from the designer UI.

4.4. Template Designer

The template designer provides all classes necessary to create and modify templates for Saddle's transformers.

The UI of the template designer consists of a tree implementation with custom nodes which each have the properties of the template. The properties are implemented as standard Java properties using a PropertySupport and custom property editors where necessary. The table used to display them is taken from NetBeans. It uses the lookup of the selected tree nodes to get their properties and display them using their editors.

This module also provides a text editor TopComponent that can be used for template related input (like in the Template Tester and Rapid Prototyper).

Apart from the UI the Template Designer registers a provider implementation into the global lookup as a means of access for other modules.

4.4.1. Template Tester

The Template Tester is a sub module of the Template Designer and provides a possibility of parsing an opened template for testing reasons. Although the template designer is able to run without the tester, the tester cannot run on its own as it takes the template information from the opened Template Designer window.

The tester is visible as an own window, by default in the top right of the screen. It will automatically provide all transformer implementations implementing the SaddleDebuggableTransformer interface and will try to find a matching transformer automatically.

4.4.2. Rapid Prototyper

The Rapid Prototyper is also a sub module of the Template Designer and can also not be used on its own. Its purpose is to give the user a possibility to rapidly create message templates from existing messages. Currently Rapid Prototypers for delimited and XML messages as well as XSD templates have been developed. The architecture allows the addition of more modules by just adding a plugin with the correct classes in the Lookup to Saddle.

Registered classes must be inserted into the layer.xml of the module and have to implement the interface lu.tudor.santec.saddle.templatedesigner.rapidprototyper.PrototypingModule in order to

become visible in the Rapid Prototyper UI. This interface provides everything including the possibility for a custom UI that is needed to set up and use a new custom rapid prototyping implementation.

5. Adding support for a new Mule version

To add support for a new Mule version first of all the new mule XSD files have to be gathered into one folder. The currently supported mule versions' XSD files can be found in the folder Mule_XSD in the root directory of the Saddle source code. It contains one subfolder for each mule version.

The config.xsdconfig file in each folder starting with 3.1 makes sure that no namespace occurs twice in the generated code later on by adding a version number to all namespaces.

When the files are now processed with scomp (one of the tools inside the XmlBeans distribution package) this generates a jar file which has to be added to a new module named muleXSDBeansXX where XX is the two digit version number of mule which has to be added to the workflow designer module as a new dependency to get access to its classes. A new SaddleMuleX.X.XComponents jar project must also be created and added to Saddle's build script (see the first few lines in build.xml) to be built and packaged together with Saddle.

After these steps have been taken the real implementation in the code of the Workflow Designer can be done. As said it is version independent where possible, but in some places it checks the mule version and handles the different versions on their own. These places can be found by looking for usages of the SupportedMuleVersions class's methods as they should have been used to determine the mule version everywhere.

6. Modifying template properties

Sometimes template properties have to be added, removed or just modified in any way. This chapter covers property modifications.

One important general piece of information is that properties are generally just written into the template XML file if their value differs from the default value. Some properties like the min- and maxoccurs properties are an exception of this rule and are always stored due to legacy code that expects them to be there in any case.

6.1. Adding a property

To add a property to the template designer the first step should be done in the Message Library by adding the new property as a constant to the StructureAttributes class. Please keep the naming schema in mind when creating the new property.

The next step should be to assign a default value to the new property in the same class. This can be done in the method getSystemDefaultValues() by adding the new property and its new value to the map of default values. The type of value given here will determine how the property is later displayed in the

Template Designer, so feel free to set any Object and don't use the String type where an Integer or Boolean value should be used.

To enable the Message Mapper to do a reverse lookup the method "getConstnameForName(String p_name)" has to get a new entry with the attribute name's value as a key and the name of the variable it is stored in as the value.

To make the property visible in the Template Designer it has to be added to the class TemplateProperties. This class provides the properties for the root node, each intermediate node (including groups) and leaf nodes of different types. These lists determine the order of the attributes as well as when they are generally applicable and visible.

6.2. Defining the display order and visibility of properties

As stated above the order is defined in the lists mentioned above. As these lists also determine when a property can be visible in general the first part of the visibility has already been defined. To have more fine grained possibilities on setting attributes visible or invisible the MsgElement has the method adjustAttributeVisibilities() which switches the visibility of some attributes based on the message type or the definition of the root node. This method makes all fine grained changes in the visibility without actually removing or adding the attributes.

6.3. Defining help texts for properties

Help texts for all attributes are stored in the AttributeInfo*.property files inside the module framework (under lu.tudor.santec.saddle.framework.resources).

In general the help texts have the following naming conventions:

"HELPTEXT_MsgAttribute_" followed by the capitalized name of the attribute itself. So for the attribute alignRight the name would be "HELPTEXT_MsgAttribute_AlignRight".

The help texts support HTML markup but do not need to be wrapped into <html> tags. To define separate help texts for the root node or non-leaf nodes the property names can be suffixed with an "R" for the root, with an "N" for non-leafs or a "G" for non-leaf nodes marked as a group.

7. Building Saddle

Creating a build of Saddle is quite straight forward. Usually there are 3 ways of doing so:

- Submit changes to the code repository
- Start Saddle in Netbeans
- Create a Saddle package

7.1. Submit changes to the code repository

As soon as any changes are committed to the code repository, a service called Jenkins will automatically create a new build of Saddle in the subsequent night. This build will be available in the Nightly Build section of the download page on the Saddle website.

7.2. Start Saddle in Netbeans

By pressing the "run" or alternatively the "run in debug mode" button, Saddle will automatically create a Saddle distribution and start it. Netbeans will use the provided Ant-scripts for building the project. This approach is the most appropriate for testing changes that have been applied to the Saddle code.

7.3. Create a Saddle package

Netbeans has the ability to generate different distributions of a package. This can be done by right-clicking on the project and choosing "Package as" in the context menu as shown in Figure 1.

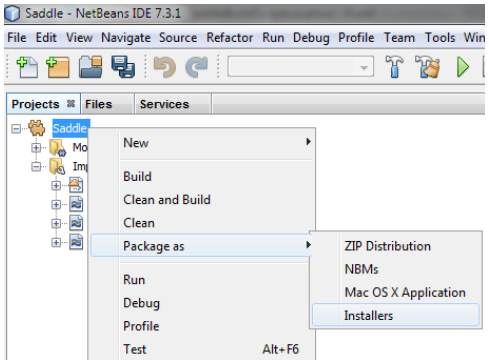

Figure 1 Context menu to create Saddle package

Due to the configuration of the Ant scripts used for creating the Saddle distribution, only "ZIP Distribution" can be used for nightly builds and release builds. For all other forms of packaging (e.g. Installers), the build script has to be configured to produce a release build.

The reason for this is the timestamp contained in the filename of the nightly build which restrains the script from finding the corresponding file when creating the package.

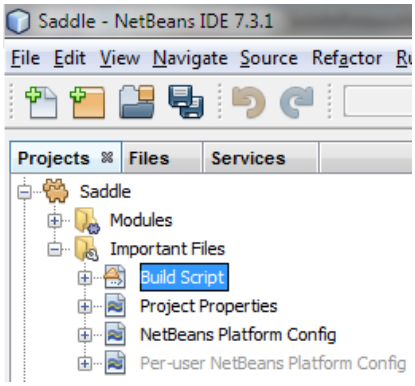

Figure 2 Location of Build Script in Netbeans

For changing between nightly build and release mode, two locations have to be manually changed in the build file. This build file is simply called "Build Script" and can be found in the "Important Files" sub directory of the project as shown in Figure 2.

The commands for both modes are already contained in the build file and just have to be activated respectively deactivated for switching between modes.

7.3.1. Setup Nightly Build mode

Figure 3 shows the command for creating the build path of the nightly build. This has to be activated in the build file. *Please assure that the command for creating a release build is deactivated.*

```
133     <!--this tag must not be active when config file is checked in to source forge otherwise it will be overwritten by Jenkins-->
134     <zip destfile="${dist.dir}/${app.name}_${timestamp}.zip">
135         <zipfileset dir="${build.launcher.dir}/bin/" filemode="755" prefix="${app.name}/bin"/>
136         <zipfileset dir="${build.launcher.dir}/etc/" prefix="${app.name}/etc"/>
137         <zipfileset dir="${temp.dir.nbexec}" filemode="755" prefix="${app.name}"/>
138         <zipfileset dir="${temp.dir.rest}" prefix="${app.name}"/>
139         <zipfileset dir="libraries" prefix="${app.name}/libraries"/>
140         <zipfileset dir="utilities" prefix="${app.name}/utilities"/>
141         <!-- Yes, the doubled app.name is a bit ugly, but better than the alternative; cf. #66441: -->
142         <zipfileset dir="${cluster}" prefix="${app.name}/${app.name}">
143             <exclude name="config/Modules/*.xml_hidden"/>
144         </zipfileset>
145     </zip>
```

Figure 3 Ant Commands for creating a Saddle nightly build

For generating the current timestamp an additional Ant command is necessary as shown in Figure 4. *Please assure that the command for creating a release build is deactivated.*

```
150   <tstamp>
151   <!--Use this property for nightly builds but be aware to comment the other timestamp property-->
152       <format property="timestamp" pattern="yyyyMMdd" />
153   </tstamp>
```

Figure 4 Ant Commands for creating an automated timestamp for the nightly build

7.3.2. Setup release mode

Figure 5 shows the command for creating the build path of the nightly build. This has to be activated in the build file. *Please assure that the command for creating a nightly build is deactivated.*

```
133     <!--this tag has to be used for builds of which zip files or installers should be created-->
134     <zip destfile="${dist.dir}/${app.name}.zip">
135         <zipfileset dir="${build.launcher.dir}/bin/" filemode="755" prefix="${app.name}/bin"/>
136         <zipfileset dir="${build.launcher.dir}/etc/" prefix="${app.name}/etc"/>
137         <zipfileset dir="${temp.dir.nbexec}" filemode="755" prefix="${app.name}"/>
138         <zipfileset dir="${temp.dir.rest}" prefix="${app.name}"/>
139         <zipfileset dir="libraries" prefix="${app.name}/libraries"/>
140         <zipfileset dir="utilities" prefix="${app.name}/utilities"/>
141         <!-- Yes, the doubled app.name is a bit ugly, but better than the alternative; cf. #66441: -->
142         <zipfileset dir="${cluster}" prefix="${app.name}/${app.name}">
143             <exclude name="config/Modules/*.xml_hidden"/>
144         </zipfileset>
145     </zip>
```

Figure 5 Ant Commands for creating a Saddle package

For each release build a version number has to be set as shown in Figure 6. *Please assure that the command for creating a nightly build is deactivated.*

```
150   <!--Use this property for release builds but be aware to comment the other timestamp property-->
151   <property name="timestamp" value="1.1.5"/>
```

Figure 6 Ant Commands for setting the Saddle version