Scribble 2.0

Developer Guide

Overview

This developers guide is intended for two groups of developers,

- those wishing to develop applications that make use of (i.e. integrate) components of the Scribble tool chain, and
- those wishing to develop additional components (or alternative implementations) for the Scribble tool chain

The document initially discusses information about the project itself, to provide background information on how issues are managed, and the build and distribution performed.

The document then discusses the architecture of the tool chain.

Finally we will look in more detail at how a validation module can be added to the tool chain.

Project Structure

2.1. Project Management

This section outlines the technology used to manage the different aspects of the project.

2.1.1. Issue Management

The issue management is handled through the JIRA system located at https://jira.jboss.org/browse/SCRIBBLE.

Issues can be created for bugs, feature requests and tasks. Bugs are used to report unexpected behaviour, and will generally be created by language/tool users. Feature requests can equally be used by users to request new language or tool features.

Tasks will only generally be created by project developers, as a way of keeping track of work that needs to be done, potentially in support of bugs or feature requests.

Issues in JIRA can be linked, where a dependency exists. It is also possible to create a simple hierarchiy with tasks, such that a parent task can contain related sub-tasks.

An important usage of the issue management system is to keep track of what issues are associated released versions of the tools, and what target release they will be implemented or fixed within. This enables users to understand the schedule of features and bug fixes, aswell as providing an automated mechanism for providing release notes describing the work associated with a particular release.

2.1.2. Project Build Management

The project build is performed using maven. A specific maven plugin, called Tycho, is used to build the Eclipse based OSGi modules.

The project also uses the Hudson continuous build and integration system to automatically trigger the build when changes are detected in the Scribble subversion repository.

Using the continuous build approach enables problems to be detected at the earliest possible stage. The build includes the execution of JUnit tests, implemented as part of the project, and the reporting of their results.

2.2. Distribution

The distribution mechanism is aimed at providing a zipped archive that contains the necessary environment.

This environment provides the ability to execute Scribble commands from the command line. However it is also possible to use the jars, contained in the `lib` and `bundle` sub-folders, directly within other Java applications.

To automatically include a new bundle in the distribution (`bundle` subfolder), it should be defined in the *org.scribble.bundles* maven group.

The build procedure also creates an Eclipse update site that includes the OSGi bundles as well as some additional Eclipse specific plugins (e.g. protocol editor).

2.3. Integration

The `runtime` branch of the project is concerned with providing integration of the OSGi bundles, defined in the `tools/bundles` branch, in different execution environments.

2.3.1. Command Line

The Scribble tools architecture is based on OSGi, which means that the OSGi compliant bundles can run within any OSGi compliant service container. However OSGi is a service framework, intended to manage services in a service container (or server).

Therefore, to leverage OSGi bundles (or services), from a command line invoked application, we need to select a specific OSGi implementation that supports this approach, as it is not defined as part of the OSGi standard.

Therefore, to provide this command line capability, we have selected the Apache Felix OSGi implementation. This is the reason that the *Felix* jars are included in the distribution's `lib` sub-folder, rather than just implementation independent OSGi jars.

Although it is possible to define new modules as part of the Scribble project, it is also possible to develop them independently and just place them within the `bundle` folder of the installed (unpacked) Scribble distribution. This will make them available as part of the command line commands (e.g. if the bundle represents an additional validation module).

2.3.2. Embedded Java

OSGi is about defining components, with well defined interfaces, and managing their isolation, to enable modules to be dyamically added or removed as necessary.

However, it is also possible to use these same components, based on the separation of interfaces and implementations, using any suitable factory or direct injection approach.

The bundles are just normal *jar* archives. They only have special significance when placed in an OSGi container.

2.3.3. Eclipse

To integrate the Scribble protocol model, parser and supporting validating modules into Eclipse, it is necessary to package them in the form of an update site. This is achieved using the maven plugin called Tycho.

The `tools` folder within the source project structure contains all of the OSGi and Eclipse based artifacts.

The `bundles` sub-folder contains the OSGi modules that can be used in an OSGi container, or integrated with the Eclipse specific plugins into an Eclipse update site.

The `features` and `plugins` sub-folders contain the more traditional Eclipse features and plugins. These plugins provide the Eclipse specific capabilities that also utilise the capabilities of the OSGi modules defined in the `bundles` sub-folder.

The `site` sub-folder provides the update site definition. This is used by Tycho to build an update site from the specified features, plugins and other OSGi bundles.

Note

The update site that is built as part of the maven build scripts is not included as part of the distribution. The update site is intended to be installed on a network server, to enable users to reference it from the update manager in their Eclipse environment.

The final sub-folder within the `tools` structure is the

tests
sub-folder. This is used to define the test plugins that are invoked as part of the Tycho build mechanism.

2.4. Modules

The `tools/bundles` sub-folder contains all of the OSGi compliant bundles that can be used in any of the integration environments.

Some of the main bundles in this sub-folder are:

1. org.scribble.common
   This module provides common capabilities used by the other bundles.

2. org.scribble.protocol
   This is the main 'protocol' related module. It contains the protocol object model, and the interfaces for the other main components in the tool chain.

3. org.scribble.protocol.parser
   This module provides the ANTLR based parser implementation.

4. org.scribble.protocol.projection
   This module provides the endpoint projection mechanism.

5. org.scribble.protocol.monitor and org.scribble.protocol.export.monitor
   These modules provide the runtime monitoring support. The export monitor bundle converts a Protocol object model into the concise monitoring finite state machine representation, and the *org.scribble.protocol.monitor* module provides the Java based monitoring engine implementation.

2.5. QA

There are two types of QA that are performed as part of the project:

1. Local test cases

Unit tests would be used to test the individual classes within the specific implementation of an interface.

2. Integration tests
   Where multiple implementations of a particular module could exist, an integration test strategy may be useful to ensure that all implementations of the same interface behaviour in the same way.

This section will discuss the second type of QA, aimed at ensuring multiple implementations behaviour in the same way.

2.5.1. Protocol Conformance Test Kit (CTK)

2.5.1.1. Parser

This part of the project structure provides a set of tests to check that the parser (being tested) processes the supplied set of test 'protocol' descriptions, and returns the correct object model.

The test protocol descriptions are stored in the `src/test/resources/tests` folder. The `global` sub-folder provides the global representation of the protocols, with the local representation of these protocols (for all of the relevant roles) being defined in the `local` sub-folder.

Each test is accompanied by a junit test, defined in the class *org.scribble.protocol.parser.ctk*. For example:

```
@org.junit.Test
public void testSingleInteraction() {
 TestJournal logger=new TestJournal();

 ProtocolModel model=CTKUtil.getModel("tests/protocol/global/SingleInteraction.spr", logger);

 assertNotNull(model);

 assertTrue(logger.getErrorCount() == 0);

 // Build expected model
 ProtocolModel expected=new ProtocolModel();

 ImportList imp=new ImportList();
 TypeImport t=new TypeImport();
 t.setName("Order");
 imp.getTypeImports().add(t);
 expected.getImports().add(imp);

 Protocol protocol=new Protocol();
 expected.setProtocol(protocol);

 protocol.setName("SingleInteraction");

 RoleList rl=new RoleList();
 Role buyer=new Role();
 buyer.setName("Buyer");
 rl.getRoles().add(buyer);
 Role seller=new Role();
 seller.setName("Seller");
 rl.getRoles().add(seller);
 protocol.getBlock().add(rl);

 Interaction interaction=new Interaction();

 MessageSignature ms=new MessageSignature();
 TypeReference tref=new TypeReference();
 tref.setName("Order");
 ms.getTypeReferences().add(tref);
 interaction.setMessageSignature(ms);
 interaction.setFromRole(buyer);
 interaction.getToRoles().add(seller);

 protocol.getBlock().add(interaction);

 CTKUtil.verify(model, expected);
}
```

The *CTKUtil.getModel()* method retrieves the protocol description from a named file, and invokes the parser implementation being tested.

The parser implementation is defined using the *scribble.protocol.parser* system property. If this property is not set, then it will default to the ANTLR based implementation.

Once the model has been retrieved using the parser, the unit test will construct an 'expected' object model.

The final step in the unit test is to invoke the *CTKUtil.verify()* method to compare the model retrieved against the 'expected' version.

To perform the verification, each model is flattened to produce a list of 'model objects'. Then the verification mechanism iterates through the list, checking that the same entry in each list is identical - first checking they are the same class, and then invoking a 'comparator' implementation for that class.

The 'comparator' implementations are defined in the *org.scribble.protocol.parser.ctk.comparators* package. The comparator implementations are registered in the static initializer for the *org.scribble.protocol.parser.ctk.ProtocolParserTest* class.

2.5.1.2. Projection

As with the parser, the CTK provides a set of tests that can be used to test the projection implementation.

The projector implementation is defined using the *scribble.protocol.projector* system property. If this property is not set, then it will use the default implementation.

The tests are performed by initially retrieving the global representation of a Protocol, and then the local representation that is associated with the particular role that will be projected. This local representation effectively becomes the 'expected' projection.

The project is then invoked, for the required role, which will produce another local representation. All that is then left to do is verify that the projected local representation is identical to the local representation loaded from the file.

An example of a projection test is shown below, where the global model is being projected to the *Buyer* role:

```
@org.junit.Test
public void testSingleInteractionAtBuyer() {
 TestJournal logger=new TestJournal();

 ProtocolModel model=CTKUtil.getModel("tests/protocol/global/SingleInteraction.spr", logger);

 assertNotNull(model);

 assertTrue(logger.getErrorCount() == 0);

  ProtocolModel  expected=CTKUtil.getModel("tests/protocol/local/SingleInteraction@Buyer.spr",
logger);

 assertNotNull(expected);

 assertTrue(logger.getErrorCount() == 0);

 // Produce projection of model to buyer
 Role role=new Role("Buyer");
 ProtocolModel projected=CTKUtil.project(model, role, logger);

 CTKUtil.verify(projected, expected);
}
```

2.5.1.3. Monitor

The monitoring CTK tests are based on simulating a set of events against the local representation of protocols, as defined in the `tests/protocol/local` sub-folder.

The JUnit tests are structured as follows:

```
@org.junit.Test
public void testSingleInteractionXSDImportAtBuyer() {
 testMonitor("tests/protocol/local/SingleInteractionXSDImport@Buyer.spr",
   "tests/monitor/SingleInteractionXSDImport@Buyer.events", false);
}
```

They simply specify the location of the local protocol representation that will be monitored, and the location of the file containing the list of events to be simulated. The final parameter indicates whether the test (or simulation) is expected to fail.

The event file has the same structure as used with the *simulate* command line function. For example,

```
receiveMessage,Order
```

```
sendChoice,validProduct
sendMessage,Order
receiveChoice,_Confirmation
receiveMessage,Confirmation
sendMessage,Confirmation
```

2.6. Samples

The `samples` sub-folder will provide samples that reflect different aspects of the Scribble notation, and the different capabilities offered by the tool chain.

Architecture

3.1. OSGi Infrastructure

The Scribble architecture is based on OSGi, to provide a means of managing the individual modules, but without causing tight coupling.

Service bundles enable implementations to be specified that implement defined interfaces. Other services can then request access to services that implement a particular interface.

The OSGi service container takes responsibility of managing the services, and providing access to requesting components.

This provides flexibility for Scribble tooling in two respects:

1. Replaceable
   The implementation of a particular Scribble interface can easily be replaced. This enables different research or industry groups to replace specific modules, with alternative implementations, while still reusing other modules.

2. Extensibility
   Some aspects of the architecture allow for multiple implementations of the same interface. Therefore, using OSGi, enables additional implementations of the same interface to be easily plugged in, without having to define any additional configuration information.

3.2. Core Components

3.2.1. Error Logging

There is a generic logging API within the Scribble framework that can be used for reporting errors, warnings, information or debuging details. This API is *org.scribble.common.logging.Journal*.

The methods generally take two parameters, a message and a property map. The message is simply a description of the issue being reported. The property map contain specific details about the issue being reported.

For example, when the parser detects a problem, it can report the nature of the problem, and provide the location of the issue in the source file.

3.2.1.1. Internationalization

To enable errors reported from the Scribble parser and validation modules, in a number of different languages, internationalization should be used.

The following code fragment provides an example of how internationalization can be achieved, using parameterised messages.

```
logger.error(java.text.MessageUtil.format(
      java.util.PropertyResourceBundle.getBundle(
       "org.scribble.protocol.Messages").getString(
       "_CHOICE_ROLE"), "from"), obj.getProperties());
```

The main message content is obtained from a properties file, with the name being supplied as the parameter to the *getBundle* method. The property file must be placed the correct package within the `src/main/resources` folder, to ensure the properties are correctly packaged by maven.

The messages within the property files can have values that include parameters. Parameters are numbered in sequential order, and defined between curly braces (e.g. {n} where 'n' is the number). For example,

```
_EXISTING_DECLARATION=Declaration already exists for name {0}
```

This message only has a single parameter.

In the previous code fragment, the *MessageUtil.format()* method takes the message as the first parameter, and a variable comma separated list of strings as the parameter values to be substituted in the message. So in the code fragment, the value *"from"* would be substituted in the *{0}* parameter of the *_CHOICE_ROLE* message, and then reported to the journal.

3.2.2. Protocol Model

The object model representation of a Protocol is defined using classes within the *org.scribble.protocol.model* package. All model classes are derived from a common *ModelObject* class, which defines common properties of all components in the model.

Where object model components are contained by another model component, we use a special list implementation called *ContainmentList*. This implementation maintains a reference to its containing parent model object, making it easier to navigate up the protocol object model hierarchy.

3.2.3. Protocol Parser

The Protocol Parser is responsible for converting the textual Scribble notation into an object model representation.

```
package org.scribble.protocol.parser;
...
public interface ProtocolParser {
 public org.scribble.protocol.model.ProtocolModel parse(java.io.InputStream is,
       org.scribble.common.logging.Journal journal);
}
```

The parser only has a single method, which takes the input stream containing the text based representation of the Scribble protocol, and a Journal for error reporting purposes.

If the Protocol has valid syntax, then a *ProtocolModel* will be returned representing the protocol in object model form.

3.2.4. Protocol Projection

The protocol projection component is used to derive a local protocol representation, associated with a nominated role, from a global protocol representation.

The interface for this component is,

```
package org.scribble.protocol.projection;
...
public interface ProtocolProjector {

 public ProtocolModel project(ProtocolModel model, Role role,
       Journal journal);
}
```

This method takes the global protocol model, the role to be projected, and a journal for reporting any errors. The result is either a local representation of the protocol model for the specified role, or null if a failure occurred.

3.2.5. Protocol Validation Manager and Validators

The validation manager, when used in a OSGi runtime environment, will listen for the activation of any implementations of the `org.scribble.protocol.validation.Validator` interface.

This means that the validation of any model can be performed using the *org.scribble.protocol.validation.ValidationManager*, rather than having to obtain instances of multiple implementations of the `Validator` interface.

When the `ValidatorManager` is used outside of an OSGi environment, it is necessary for the validators to be added to the manager by other means.

3.2.5.1. Model Component based Validation

One of the default validation implementations is *org.scribble.protocol.validation.rules.DefaultProtocolComponentValidator*. This class is derived from a generic based class that validates supplied protocol models by visiting the component objects within the model, and invoking a specific 'validation rule' based on the type of the model object.

This default implementation is used to provide the basic validation rules for the model components. For example, to ensure that the roles defined within an interaction have been previously declared within the scope containing the interaction.

3.2.6. Exporting the Protocol model to other representations

The Scribble tools provide a mechanism for exporting a Scribble Protocol object model to other representations. This module has the following interface,

```
package org.scribble.protocol.exporter;
...
public interface ProtocolExporter {
 public String getId();
 public String getName();
 public void export(ProtocolModel model, Journal journal, java.io.OutputStream os);
}
```

Each 'exporter' implementation defines an id and more descriptive. The id can be used to lookup the implementation from the *ProtocolExportManager*, whereas the name can be used as a descriptive name for display to users.

The *export* method takes the protocol model to be exported, the journal where to report errors, and the output stream which will be the destination for the exported representation.

The *org.scribble.protocol* bundle contains a default exporter to convert the Scribble object model representation into a text based representation. The 'id' for this implementation is *txt*.

3.2.6.1. Monitor

Another export module is the *org.scribble.protocol.export.monitor*.

This implementation produces an XML based finite state machine representation of the protocol, for use by the Scribble Protocol Monitor.

3.2.7. Scribble Protocol Monitor

The Scribble Protocol Monitor provides a runtime component that can observe messages being sent and received by an endpoint application, assuming a particular role within a protocol, and ensure that it conforms to the expected behaviour.

Warning

TO BE DOCUMENTED

3.3. Command Line Actions

The first step is to define the command implementation of the *org.scribble.command.Command* interface. This can be created in the *org.scribble.command* Eclipse plugin.

To initialise the command, as part of an OSGi runtime environment, the command implementation can be instantiated in the *org.scribble.command* plugin's *Activator*, and then registered with the bundle context.

If the command requires other OSGi services, then these can be established by setting up service listeners for the relevant service interface classes. When OSGi services are registered, then the relationship can be established.

This command mechanism will generally only be used as part of the command line approach, and therefore does not need to be initialised in other ways. However other dependency injection techniques could be used if appropriate.

The only remaining step is to create the scripts necessary to enable a user to invoke the command. This can be achieved by copying one of the existing scripts, in the `distribution/src/main/release` folder (such as `parse.sh`), and modify the parameter details as necessary.

Note

The first parameter of the Java application, invoked within the script, must be the name of the command. This must match the value returned by the *getName()* method on the command interface.

Developing a Validator

This section will describe how to create a validator as part of the Scribble project structure. The same approach can also be used to create a validator module outside the scope of the Scribble project.

To explain how to create a validator, we will use the 'connectedness' validator as an example.

4.1. Create the Validator OSGi bundle

This section will explain how to create the OSGi bundle, for the validator, from within the Eclipse environment.

The first step is to create the plugin project, using the *New->Project->Plugin Project* menu item. When the dialog window is displayed, uncheck the "Use default location" checkbox, and browse to find the appropriate location for the new project.

For this project, the location will be the `tools/bundles/org.scribble.validator.connectedness` folder within the Scribble project structure. It will be necessary to create the folder for the *org.scribble.validator.connectednesss* part of the location - the folder being named after the OSGi bundle identity.

Ensure the 'Create java project' checkbox is ticked, and then set the source folder to be `src/main/java` and set the *Target Platform* to a standard 'OSGi Framework'.

Then press the *Next* button to set some details associated with the plugin, such as the version, description, provider, etc.

In this example, we will be registering the validator using the OSGi *registerService* method. This is performed in the bundle activator, whose class is set in the plugin details. For example, in the start method of the created Activator, we would have:

```
public void start(BundleContext context) throws Exception {
       Properties props = new Properties();

       ProtocolValidator validator=new ConnectednessValidator();

       context.registerService(ProtocolValidator.class.getName(),
   validator, props);
}
```

4.2. Establish Bundle Dependencies

Depending on the type of bundle being developed, it may have a different set of dependencies than the ones required by this 'connectedness' validator. However the configuration approach will be the same.

Go to the `META-INF/MANIFEST.MF` file and select it. This will cause the plugin manifest editor to be displayed.

Select the *Dependencies* tab and select the other bundles that will be required, or preferrably select the packages to be imported (as this avoids dependency on specific bundles, and instead just identifies the packages required). For this example validator, we just need to add the packages from the *org.scribble.common* bundle which is used by all Scribble plugins. However if additional packages were required, then they could be added as imported packages.

4.3. Implement the Module

Each module will be different, and therefore discussing specific implementation details will not be possible.

However validation modules will tend to access the complete model, but possibly only be interested in certain parts of it. Therefore usually the validation modules will define an implementation of the `org.scribble.protocol.model.Visitor` interface.

The actual main class within the validator module would implement the `org.scribble.protocol.validation.ProtocolValidator` interface.

There may also exist specialised implementations of the *ProtocolValidator* interface that help support the validation process. For example, the *ProtocolComponentValidator* which triggers a *ProtocolComponentValidatorRule* based on the type of the model component. The visitor is used to traverse the model to identify the model components being validated. So its possible, if validation of only a couple of model component types is required, to derive a specialisation of the *ProtocolComponentValidator* class with the relevant rule implementations.

4.4. Implement the Tests

Tests can be implemented in two ways, depending upon the nature of the bundle.

If the bundle is representing an implementation of a common interface, where the result returned from the bundle is the key point, then integration tests associated with the interface can be useful.

For example, there is a special bundle used to provide a *Conformance Test Kit* for the protocol parser, projector and monitor. This means that the same set of integration tests can be used regardless of the implementation of those components being used.

The other set of tests that are useful are specific to the bundle implementation. In standard Java plugins, these tests will usually be provided within the bundle itself, by defining the JUnit test classes within the `src/java` location, with any required resources being placed in `src/resources`.

However using Tycho to build the plugins and OSGi bundles means that it is better to locate these bundle implementation specific tests in a companion *test* plugin located in the `tools/tests` sub-folder.

The next step is to create the JUnit test. First create the appropriate package within the `src/java` location.

Then select the *New->Other->JUnit Test Case* menu item associated with the package. When this is first performed, you will be asked which version of JUnit should be used. Select the *New JUnit 4* radio button.

After pressing the *Next* button, you will be asked about adding JUnit to the classpath.

Choose the *Open the build path property page* option and press the *Ok* button. The reason for not adding JUnit directly, is that this would cause it to be included in the list of bundle dependencies in the OSGi manifest, which would mean that the runtime environment that includes the validator would also have a dependency on JUnit.

Therefore we need to add the JUnit jars to the Eclipse project in a different way. In the build path dialog, select the *Libraries* tab and then select the *Add Library* button. When the list of libraries is presented, select the *JUnit* entry and press *Next* where the JUnit version should be set to *JUnit4* and then press the *Finish* button.

4.5. Create the Maven POM

We need to create a `pom.xml` for both the main plugin (or OSGi bundle) and the test plugin.

The best approach is to copy the `pom.xml` file from one of the other modules, and simply update the relevant sections (e.g. artifact id, description and dependencies).

Once the `pom.xml` for the module has been defined, it needs to be linked into the `pom.xml` of its parent. This is `tools/bundles` for the main bundle, and `tools/tests` for the test plugins.