Scribble Java
Developer Guide

Chapter 1. Protocol

This section explains how to make use of the Scribble Java tools to parse, validate and project a Scribble protocol. The following sections of this document explain how applications can then make use of these validated (and potentially projected) protocols to perform further tasks, such as monitoring message exchanges to ensure they conform to a defined protocol, or simulating message traces against endpoint simulators.

The Java tools make use of maven to store its artifacts (i.e. jars). These are associated with the group id *org.scribble* and the artifact id *scribble-<component>*, where the <component> is the individual area represented by the artifact. As well as there being an artifact per component of the tooling, there is an additional *scribble-core* artifact that contains items shared by all of the components.

1.1. Common Components (maven artifact id *scribble-core*)

This artifact contains general interfaces/classes for issue logging, context, resources and the protocol model (local and global variants).

1.1.1. Issue Logging

Whenever a component needs to perform processing on the protocol, to identify parsing, validation or projection issues, then the component will use the *org.scribble.logging.IssueLogger* to report any errors, warnings or other general information.

As part of the core artifact there is a *org.scribble.logging.ConsoleIssueLogger* implementation that reports any issues to the console, however an application is free to provide its own implementation. For example, the Eclipse tooling contains an implementation that converts the issues into Eclipse markers for reporting in the *Marker* or *Problems* views.

1.1.2. Context

The *org.scribble.context* package contains the following components that may be used with various protocol processing capabilities:

- Module Loader

The *org.scribble.context.ModuleLoader* interface is used during a variety of processing stages, e.g. parsing, validation, monitoring, etc. It is responsible for retrieving a *org.scribble.model.Module* object model associated with a fully qualified module name.

There is a default implementation of the module loader *org.scribble.context.DefaultModuleLoader* that simply provides a caching capability for loaded modules. It is expected that a derived loader implementation will be provided, that will leverage the caching capability of the default

implementation, but will provide the environment specific knowledge of how to obtain the modules. One such concrete implementation is *org.scribble.parser.ProtocolModuleLoader* which can be found in the *scribble-parser* component.

- Module Context

The module context is responsible for providing support services to any processing that is occuring on a particular module. The focus of a particular module context instance will be an individual *org.scribble.model.Module* instance. Based on the definitions contained within that module instance, an application can request access to members, either within that module, or in an associated module (identified by a fully qualified name).

1.1.3. Resources

Within the *org.scribble.resources* package is contained classes/interfaces to provide support for locating and loading resources. These capabilities can be used to load modules, as well as any other appropriate resources used during parsing, validation or further stages.

1.1.4. Protocol Model

The model contains the general module components, as well as the specfic components to represent the local and global variations of the Scribble protocol.

The top level model component is *org.scribble.model.Module*.

1.2. Parsing (artifact id *scribble-parser*)

The parser is the component responsible for taking a text based description of a Scribble protocol and transforming it into an object model. As part of this process, it will verify that the syntax of the protocol description is valid, and report any errors using the supplied *org.scribble.logging.IssueLogger*.

```
String path=....;  // Colon separate directory paths where scribble modules
 are located
java.io.InputStream is=....; // Input stream containing text description of
 scribble protocol

org.scribble.parser.ProtocolParser pp=new
 org.scribble.parser.ProtocolParser();
org.scribble.logging.IssueLogger logger=new
 org.scribble.logging.ConsoleIssueLogger();
org.scribble.resources.DirectoryResourceLocator locator=new
 org.scribble.resources.DirectoryResourceLocator(path);

org.scribble.context.ModuleLoader loader=new
 org.scribble.parser.ProtocolModuleLoader(pp, locator, logger);
```

```
org.scribble.resources.Resource res=new
 org.scribble.resources.InputStreamResource(path, is);

org.scribble.model.Module module=pp.parse(res, loader, logger);
```

The last line of this example shows the parser being involved. It takes three parameters:

- the resource, containing the text based scribble protocol description
- the loader, to load any additional modules (or potentially other resources) that may be required to support the parsing of the module
- the logger, to report any issues that arise from parsing the protocol description

If the parser returns a module, then it means that it was successfully parsed. Otherwise the syntax errors will be reported to the issue logger and no module will be returned.

1.3. Validating (artifact id *scribble-validation*)

The validator is the component responsible for evaluating a protocol module (*org.scribble.model.Module*) to determine if it conforms to a set of predefined rules (e.g. wellformedness conditions). As with the parser, any issues will be reported to the supplied *org.scribble.logging.IssueLogger*.

```
org.scribble.logging.IssueLogger logger=...;
org.scribble.resources.Resource res=...;
org.scribble.context.ModuleLoader loader=...;
org.scribble.model.Module module=...;

org.scribble.context.ModuleContext context=new
 org.scribble.context.DefaultModuleContext(res, module, loader);

org.scribble.validation.ProtocolValidator pv=new
 org.scribble.validation.ProtocolValidator();

pv.validate(context, module, logger);
```

Most of the components used in this example validation were introduced in the parser section above. The new components in this example are the *ProtocolValidator*, which will perform the validation, and the *ModuleContext*. As discussed in a previous section, the module context provides access to members (e.g. type or protocol definitions) in a particular module, or associated module.

1.4. Projection (artifact id *scribble-projection*)

In the context of Scribble, projection is the term used to describe extracting the local endpoint behaviour of a role defined within a global protocol. The global protocol describes the interactions

between multiple parties, whereas the local protocol described the interactions from a particular role's perspective.

Being able to filter out just the responsibilies of an individual role, from the potentially complex set of interactions that may be defined in a global protocol between many participants, is important - primarily for being able to determine whether an implementation of that role (endpoint) is statically or dynamically conforming to the expected behaviour.

```
org.scribble.logging.IssueLogger logger=...;
org.scribble.resources.Resource res=...;
org.scribble.context.ModuleLoader loader=...;
org.scribble.model.Module module=...;

org.scribble.context.ModuleContext context=new
 org.scribble.context.DefaultModuleContext(res, module, loader);

org.scribble.projection.ProtocolProjector projector=new
 org.scribble.projection.ProtocolProjector();

java.util.Set<Module> projected=projector.project(context, module, logger);
```

The code is very similar to the validation example, with the exception that we are creating a *ProtocolProjector* and the projection results in a set of modules representing the local protocol definitions.

Chapter 2. Monitor

The monitoring capability is used to ensure that a system conforms to a protocol description at runtime. This is a form of dynamic validation, or conformance checking.

2.1. Converting a Protocol into a Monitor State Machine

To efficiently monitor a running system, to ensure that it is conformed to one or more roles within a protocol description, it is necessary to transform the text based description (and even the object model representation) into a form that can more effectively be used to drive a runtime monitoring solution.

```
org.scribble.context.ModuleLoader loader=...;
org.scribble.resources.Resource res=...;

org.scribble.model.local.LProtocolDefinition lp=...; // Obtain the required
 local protocol definition

org.scribble.monitor.export.MonitorExporter exporter=new
 org.scribble.monitor.export.MonitorExporter();

org.scribble.context.ModuleContext context=new
 org.scribble.context.DefaultModuleContext(res, lp.getModule(), loader);

org.scribble.monitor.model.SessionType type=exporter.export(context, lp);
```

The first step is to obtain the module that contains the local protocol definition to be monitored. This can ether be achieved by parsing a textual representation of a local protocol definition, or by projecting the local modules from a global module.

Once the module is obtained, then the specific local protocol definition can be retrieved. As a module may contain multiple local protocol definitions, it is important to select the one that represents the initial (or top level) protocol definition from the perspective of what needs to be monitored.

Once the exporter has been instantiated, invoke the *export* method with the selected local protocol definition. This will export the protocol definition into a state machine representation associated with the returned *org.scribble.monitor.model.SessionType* object. This object will be used in subsequent runtime monitoring session instances to define the behavioural type being verified.

2.2. Dynamically Monitoring Message Exchanges

2.2.1. Managing session instances

It is currently out of the scope of the Scribble monitor to manage session instances. It is up to the application invoking the monitor to determine:

- When a new session instance must be created and initialized

In this situation, the application should instantiated an instance of the *org.scribble.monitor.SessionInstance* class and supply it, along with the relevant *org.scribble.monitor.model.SessionType* object (defining the behavioural type to be monitored), to the *initializeInstance* method of the monitor, e.g.

```
org.scribble.monitor.Monitor monitor=new
 org.scribble.monitor.DefaultMonitor();
org.scribble.monitor.model.SessionType sessionType=....;

org.scribble.monitor.SessionInstance instance=new
 org.scribble.monitor.SessionInstance();

monitor.initializeInstance(sessionType, instance);
```

The new session instance should then be stored by the application, associated with some relevant key that can be used to retrieve it later.

- When an existing session instance should be retrieved

If a key is obtained from the interaction being monitored, possibly by extracting relevant information from the message content or header, then it can be used to locate an existing session instance.

- When a session instance is no longer required

The *org.scribble.monitor.SessionInstance* class has a method called *hasCompleted* which will return a boolean result, indicating whether the session instance has completed.

This should be checked after any processing of the session instance by the Scribble monitor. If this method returns *true*, then the session instance object should be removed from the set of application managed session instances.

2.2.2. Verifying behaviour

When behaviour is detected, and an appropriate session instance object created or retrieved, then the behaviour can be verified using the Scribble monitor. Currently the following types of verification can be performed:

- Message Sent

The following is an example of how to verify a sent message:

```
org.scribble.monitor.Monitor monitor=....;
org.scribble.monitor.model.SessionType sessionType=....;
org.scribble.monitor.SessionInstance instance=....;
```

```
String toRole=....;
org.scribble.monitor.Message mesg=new org.scribble.monitor.Message();

mesg.setOperator("placeOrder");
mesg.getTypes().add("{http://acme.org/ordermgmt}Order");
mesg.getValues().add("<order xmlns=\"http://acme.org/ordermgmt\" id=\"xyz
\" />");

boolean result=monitor.sent(sessionType, instance, mesg, toRole);
```

The first lines are simply present to identify the types associated with the parameters to the *sent* method.

The next block would identify the *toRole*, i.e. the role that the message is being sent to, and the message details. The message includes an operator name, and a list of parameter types/values.

Note

Currently the values are not used, so it is not necessary to supply them, but in the future they will be used in the evaluation of assertions.

The monitor is then invoked using the *sent* method, supplying the session type and instance, as well as the message and *to* role. The result of this method is a boolean value indicating whether the monitor considered it to be valid or not.

- Message Received

The following is an example of how to verify a received message:

```
org.scribble.monitor.Monitor monitor=....;
org.scribble.monitor.model.SessionType sessionType=....;
org.scribble.monitor.SessionInstance instance=....;

String fromRole=....;
org.scribble.monitor.Message mesg=new org.scribble.monitor.Message();

mesg.setOperator("placeOrder");
mesg.getTypes().add("{http://acme.org/ordermgmt}Order");
mesg.getValues().add("<order xmlns=\"http://acme.org/ordermgmt\" id=\"xyz
\" />");

boolean result=monitor.received(sessionType, instance, mesg, fromRole);
```

The first lines are simply present to identify the types associated with the parameters to the *received* method.

The next block would identify the *fromRole*, i.e. the role that the message is been received from, and the message details. The message includes an operator name, and a list of parameter types/values.

Note

Currently the values are not used, so it is not necessary to supply them, but in the future they will be used in the evaluation of assertions.

The monitor is then invoked using the *received* method, supplying the session type and instance, as well as the message and *from* role. The result of this method is a boolean value indicating whether the monitor considered it to be valid or not.

Chapter 3. Simulation

Simulation is performed by defining a *trace* file, containing a sequence of actions (e.g. message transfers), and one or more simulation definitions identifying how each role should be simulated.

3.1. Defining a Trace

The *trace* model can be serialized as a JSON representation, e.g.

```
{
 "name":"RequestResponse-1",
 "steps":[{
  "type":"MessageTransfer",
  "message":{
   "operator":"buy",
   "types":["{http://scribble.org/example}OrderRequest"],
   "values":[""]
  },
  "fromRole":"Buyer",
  "toRoles":["Seller"]
 },{
  "type":"MessageTransfer",
  "message":{
   "operator":"buy",
   "types":["{http://scribble.org/example}OrderResponse"],
   "values":[""]
  },
  "fromRole":"Seller",
  "toRoles":["Buyer"]
 }],
 "simulations":[{
  "roleSimulators":{
   "Buyer":{
    "type":"MonitorRoleSimulator",
    "module":"scribble.examples.RequestResponse",
    "role":"Buyer",
    "protocol":"First"
   },
   "Seller":{
    "type":"MonitorRoleSimulator",
    "module":"scribble.examples.RequestResponse",
    "role":"Seller",
    "protocol":"First"
   }
  }
 }]
}
```

In this example trace file, two steps (or actions) are defined. The first is representing the order request being sent from the *Buyer* role to the *Seller* role. The second is representing an order response being returned from the *Seller* role to the *Buyer* role. The *message* component defined the operator, list of parameter types, and list of parameter values. The values are currently optional - however once assertions are supported, the value will need to be provided.

The *simulations* section defines a list of simulations. Each simulation defined an optional name, and a map of role names to role simulators.

In this example, the only role simulator type used is **MonitorRoleSimulator** which uses the Scribble monitor to verify that the message transfers defined in the trace conform to the Scribble protocol identified by the module, role and protocol.

Note

If a trace contains steps associated with roles that are not defined within the role simulator map, then those roles will be ignored when performing the simulation.

3.2. Performing a Simulation

To perform a simulation from within your own application, you need to perform the following steps:

```
// Create a locator that can be used to load the scribble modules and any
 associated resources
org.scribble.resources.ResourceLocator locator=new
 org.scribble.resources.DirectoryResourceLocator(...);

// Build or load the trace (e.g. deserialize the JSON representation)
org.scribble.trace.model.Trace trace=...;

// Create a context using the locator
org.scribble.trace.simulation.SimulatorContext context=new
 org.scribble.trace.simulation.DefaultSimulatorContext(locator);

// Create the simulator
org.scribble.trace.simulation.Simulator simulator=new
 org.scribble.trace.simulation.Simulator();

// Instantiate an implementation of the listener interface, to be informed
 when steps are simulated
// successfully or unsuccessfully
org.scribble.trace.simulation.SimulationListener l=...;

// Add the listener to the simulator
simulator.addSimulationListener(l);

try {
```

```
 // Simulate the trace
 simulator.simulate(context, trace);
} catch (Exception e) {
 ...
}

// Unregister the listener
simulator.removeSimulationListener(l);
```