Extending Tuscany by contributing a new implementation/binding type

Raymond Feng, Jeremy Boynes

Abstract

The SCA specification defines an extension model which allows the assembly model to be extended with support for new interface types, implementation types and binding types. The Tuscany runtime is designed to provide such extensibilities. In this tutorial, we will first discuss the key extensible aspects in the architecture and then provide a step-by-step walk through to illustrate how to contribute a new component implementation type using the JavaScript as an example.

1. Overview

We wanted to make sure that Tuscany could be extended in simple but flexible way. To that end, we based the extension model on the SCA Assembly Model itself and allow extensions to be contributed as Module Fragments. Each fragment contains an XML file containing part of an SCA assembly which can be used to define components and wire them together. This guide assumes familiarity with the SCA Assembly Model.

To separate extension code from application code, we introduced a "system" implementation type for components. This "system" type supports Java components in a way that is similar to the Java programming model from the specification but with a few additional privileges that let them become part of the runtime.

To extend Tuscany, you simply add system components into the assembly that defines the runtime. This adds functionality, functionality that can be used to support a new programming language for components, new ways of communicating in and out of the system, new services that can be made available to applications, anything really ...

In practice, there are a few well defined types of extension that people typically want to add:

- New types of implementations for application components such as JavaScript, BPEL and Spring
- New types of network transport such as HTTP and JMS
- New types of network protocol such as SOAP and JSONRPC
- New types of data binding such as SDO and JAXB
- New services that can be provided to applications such as a database connection pool

Many of these extensions need to integrate into the process that is used to deploy application components. For example, when a user uses a new implementation type in their assembly file, the extension that provides that new implementation needs to be activated during deployment so that it can create the component they defined.

More detailed information on the general deployment process can be found later in this document, but in brief there are three touchpoints where extensions need to interact with it:

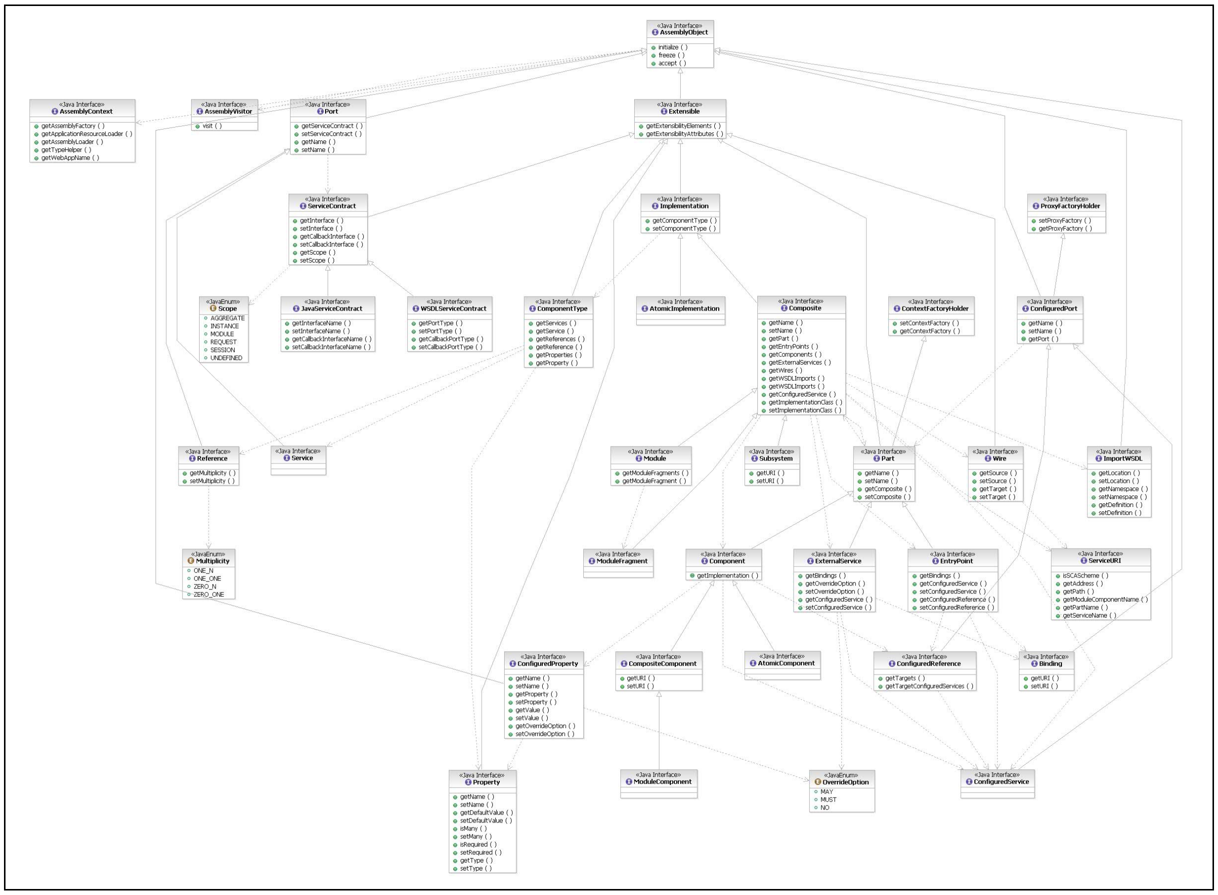
- Contributing a component that can extract configuration information from an XML stream
- Contributing a component that can build contexts that represent application components
- Contributing a component that helps wire application compnents together

2. Configuration Model

Internally, Tuscany uses a configuration model to represent the complete definition of an application that will be run. This model is loaded from various persistent artifacts (such as XML files, other configuration files, annotated Java classes, etc.) and used by the builders to create the actual runtime contexts that comprise the running application.

The configuration model is based on the SCA assembly model with additions for Tuscany-specific features and for each of the installed extensions. It should be stressed that although there is a natural similarity to the XML data model defined by the specification the configuration model is driven by the needs of the Tuscany implementation and its extensions.

The model can be described by the following UML:



The code for the core model is contained in the "model" module located at http://svn.apache.org/repos/asf/incubator/tuscany/java/sca/model and all model object derive from the core AssemblyObject interface. We use interfaces to describe most model items and provide a factory that can be used to create instances; this is optional and it is possible (if not recommended) for extensions to add to the model using simple JavaBeans.

For a full description of the model please see the JavaDoc. Some common extension points in the model are:

- Implementation interface that defines the actual implementation that should be used for a component. This would typically be extended by new component implementation types to describe the physical implementation that should be used (such as a Java class name or a script file).
- Binding interface that defines a binding to be applied to an EntryPoint or ExternalService. This would typically be extended by new transport or protocol bindings to describe how invocations will be represented on the wire (such as a web-service or IIOP call).

3. Deployment Process

The SCA Assembly model provides a very flexible way for users to define the structure of their applications. However, the specification is still in the process of defining how these assemblies should be deployed to physical servers. Further the Assembly specification itself is still developing.

In light of that, we have tried to develop a very flexible solution for deployment in Tuscany that allows us to modularize the process and react to evolutionary (and even revolutionary) changes in the assembly and deployment models. This solution assumes that as an application is deployed its configuration goes through three different forms:

1. Persistent artifacts that have been provided by the user. These may be simple XML files, they may be bundles of code (e.g. in JAR or CAB files), they may be values in some global configuration system. When a user deploys an application, Tuscany needs to locate and load all these different artifacts and assemble them together into a consistent and complete desciption of what the user wants to run.
2. The complete model of the application created in memory. This is an intermediate form created from the persistent artifacts which describes the application - we call it the logical configuration model. It is designed to consistently represent the application. This model is described in ConfigurationModel.
3. A set of runtime contexts that actually run the application. These are optimized and wired together in a way that maximises the performance of the system. The emphasis is on short code paths that can be carefully tested so that we increase the stability of a production server.

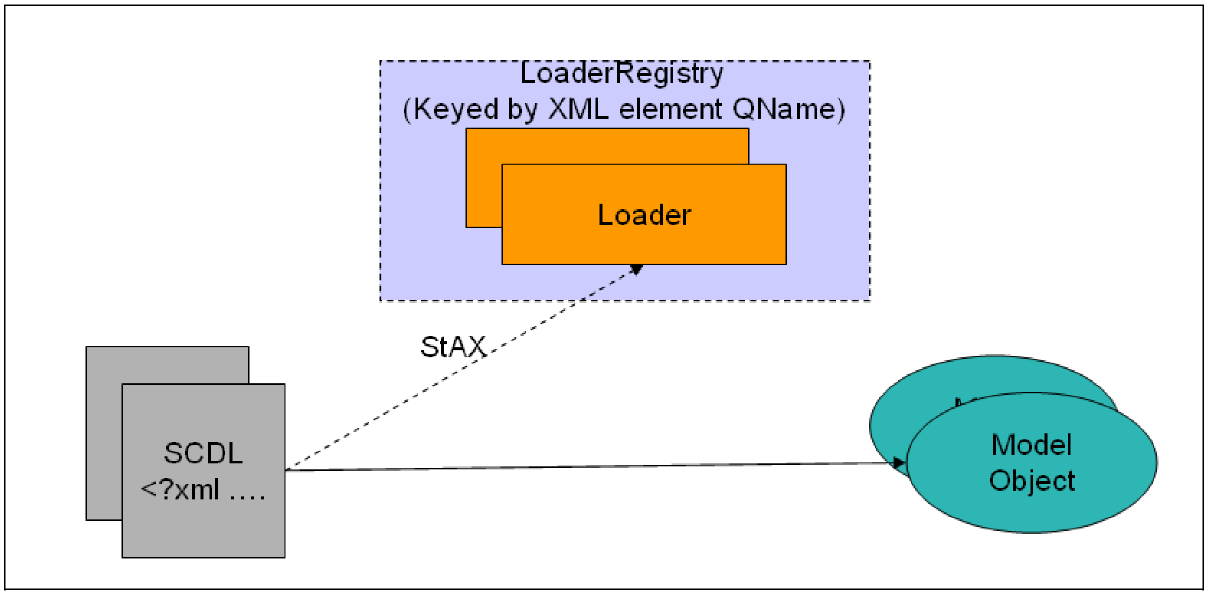
We call the process of gathering together the persistent artifacts and building the logical configuration model "loading" as the primary activity is reading files, parsing them and building a consistent view. We call the process of converting the configuration model into runtime contexts "building" as the primary activity is building and optimizing runtime structures from the description in the model.

We have tried to keep a very clear separation between the two activities. There may be many ways in which persistent configuration information is stored and we may well need to support several concurrently. For example, in a clustered environment a management node may load the configuration from some central location but each of the worker nodes may just copy the configuration from it. Or, users may want to configure the system in different ways, for example using Groovy script instead of XML; we make use of this in our test environment by constructing the model programatically as part of the test suite.

3.1 Loading Phase

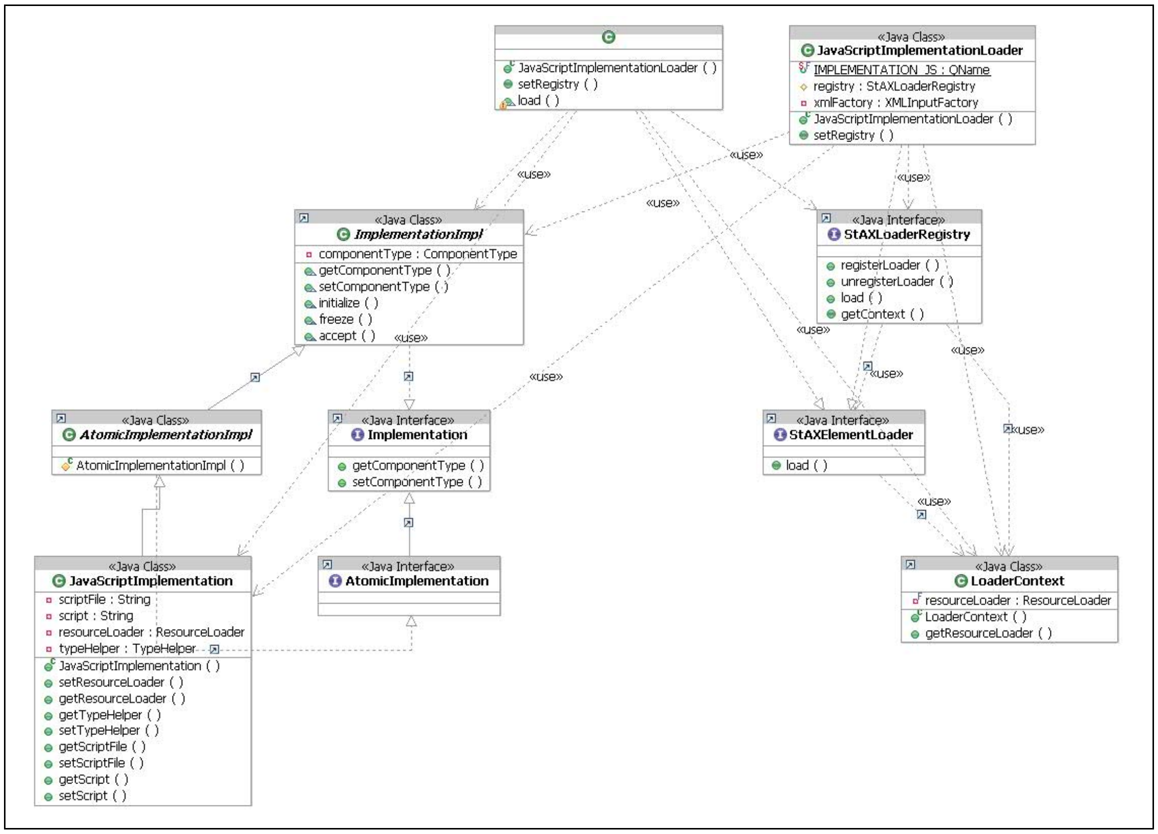
The XML used for the SCA Assembly model is structurally quite simple but very extensible; almost every element support extension elements and the specification itself uses substitution groups to allow specific types of implementation and binding to replace/extend key elements. Further, the model itself makes reference to artifacts located outside the XML instance (such as WSDL definitions, Java classes and other SCA artifacts like componentType sidefiles) that are needed to build up the complete representation of an application.

To handle this extensibility and the external artifacts, in Tuscany we have chose to use a streaming approach to parsing based on the Java StAX framework. Rather than load XML artifacts into objects using a data binding technology such as SDO or JAXB, the loader framework provides a mechanism for extension to register their willingness to parse any XML element. As the XML file is read, elements are dispatched to registered loaders who can then handle the stream in any way they choose. As part of processing the stream, they can also read any associated artifacts especially those that may be needed later during the configuration loading process (such as WSDL definitions).



To partipate in the loading process, an extension should provide a component that implements the StAXElementLoader interface and should register that component with the framework's StAXLoaderRegistry during initialization.

The players in the loading phase are captured in the following UML class diagram.

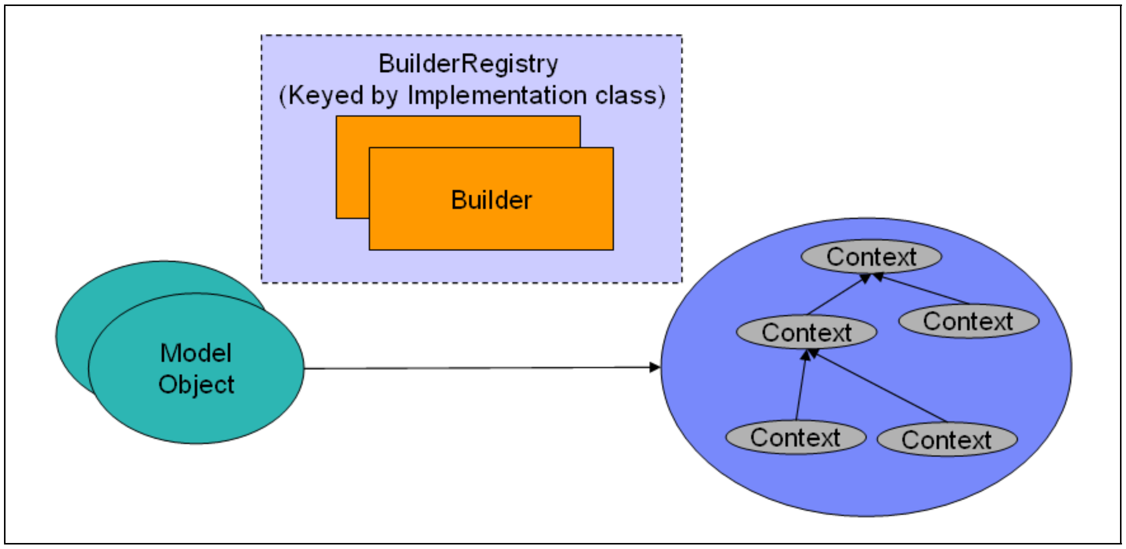


3.2 Building Phase

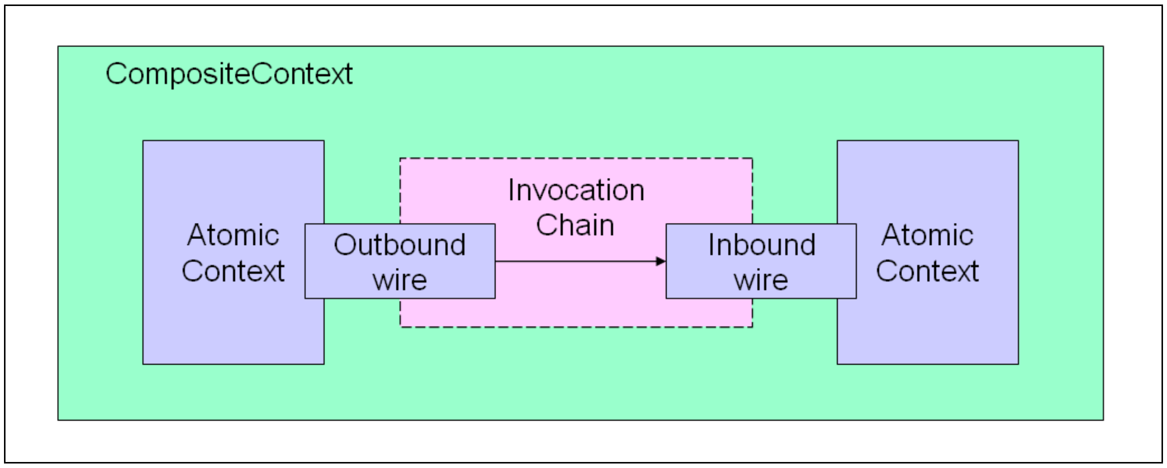
The builders are responsible to build runtime context from the model objects to represent application components. Context is the runtime peer to the model object.

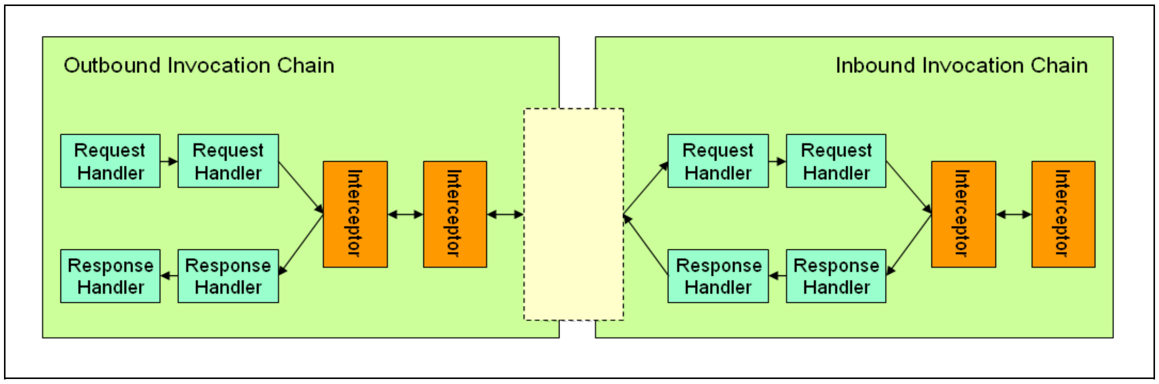
The building phase can itself be subdivided into two phases:

❖ Context building, where the runtime context for each component implementation is built. This allows component implementations to set up the context for a component in isolation. The output is a "hairy" context containing the implementation of the component itself with a set of stubs (the hairs) that need to be connected to other components.

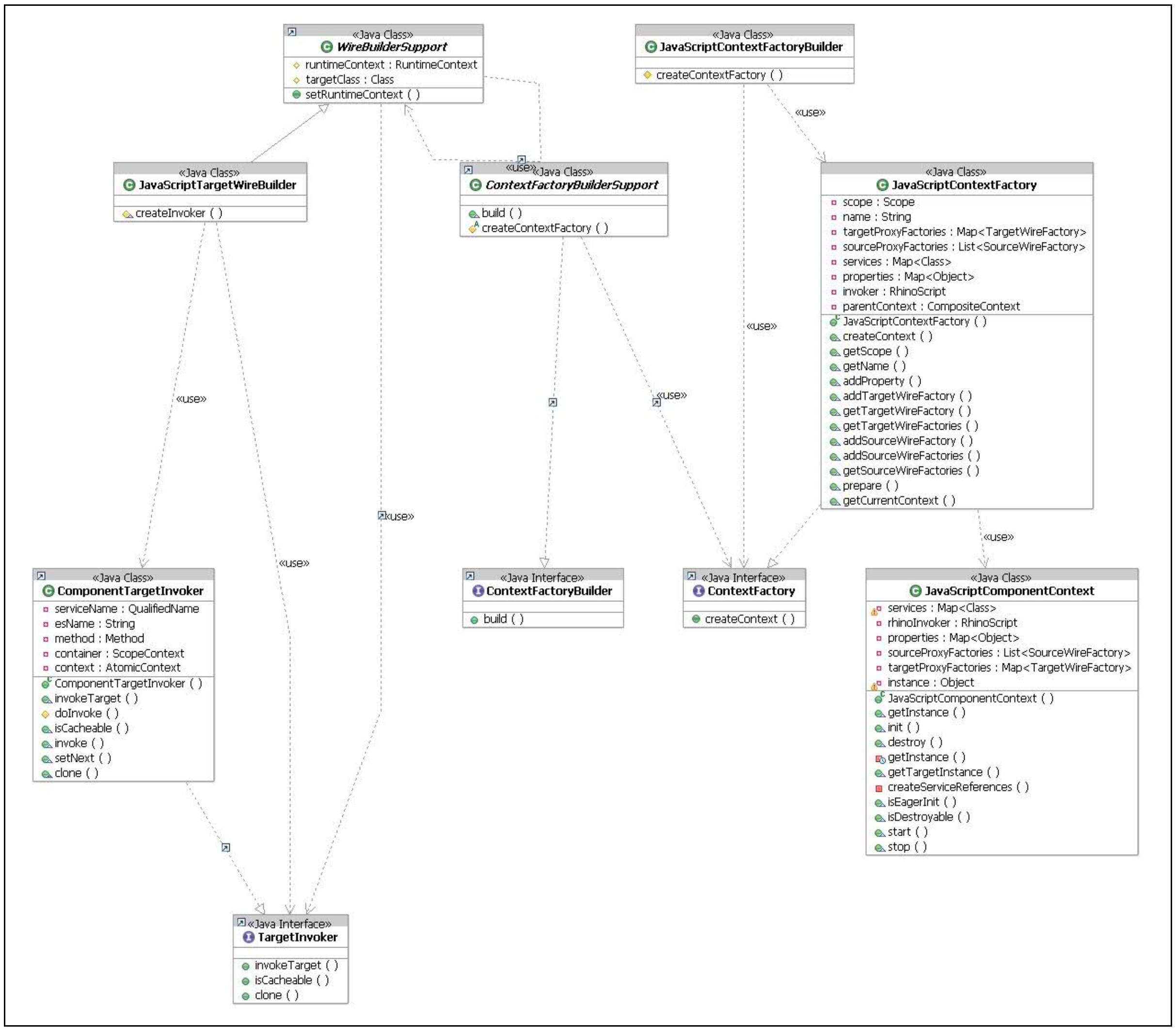


- Wire building, where the wires that connect components are created. This involves connecting the "hairs" from each isolated component to each other to create a fully connected runtime. Wire builders are reponsible for finding the most optimized connection between two components and for making sure all the policies needed by the service contract are present.





The players in the loading phase are captured in the following UML class diagram.



4. Contribute a new component implementation/binding type

In this section, we guide you through all the steps required to add a new component implementation type to Tuscany. We'll use the XML metadata files and java code from the Tuscany Rhino container (JavaScript).

4.1 Define the SCDL extensions for your component implementation

As illustrated below, you'll extend the base SCDL to provide the metadata configuration for your new type. In the sample case, we have a new implementation represented by the "implementation.js" element under namespace "http://org.apache.tuscany/xmlns/js/0.9".

```
<module xmlns=http://www.osoa.org/xmlns/sca/0.9
        xmlns:js="http://org.apache.tuscany/xmlns/js/0.9"
        name="JavaScriptTests">

    <component name="HelloWorldComponent1">
        <js:implementation.js scriptFile="tests/HelloWorldImpl1.js"/>
    </component>
</module>
```

4.2 Create the model object to represent your extensibility elements/types

JavaScriptImplementation

```
public class JavaScriptImplementation extends
AtomicImplementationImpl {

private String scriptFile;
...


    public JavaScriptImplementation() {
        super();
    }

       ...
    public String getScriptFile() {
        return scriptFile;
    }

    public void setScriptFile(String fn) {
        scriptFile = fn;
    }

}
```

4.3 Create a StAX loader to extract metadata from the XML stream into the model object

Using the JavaScript implementation type as an example, this would be something like:

```
@Scope("MODULE")
public class JavaScriptImplementationLoader implements
StAXElementLoader<JavaScriptImplementation> {
    public static final QName IMPLEMENTATION_JS = new
QName("http://org.apache.tuscany/xmlns/js/0.9", "implementation.js");

    private StAXLoaderRegistry registry;

    @Autowire
    public void setRegistry(StAXLoaderRegistry registry) {
        this.registry = registry;
    }

    @Init(eager = true)
    public void start() {
        registry.registerLoader(IMPLEMENTATION_JS, this);
    }

    @Destroy
    public void stop() {
        registry.unregisterLoader(IMPLEMENTATION_JS, this);
    }
```

In this example, an instance of this component would be included with the extension. When the extension module starts, the component is initialized immediately due to the presence of the `@Init(eager = true)` annotation. In its init method, it registers itself with the loader registry as a loader for elements with the QName `<implementation.js>` in the JavaScript extension's namespace.

As XML configuration files are being parsed, when the loader sees the registered element it will call this component's load method to handle it. Continuing the JavaScript example, we see:

```
    public JavaScriptImplementation load(XMLStreamReader reader,
ResourceLoader resourceLoader) throws XMLStreamException,
ConfigurationLoadException {
        String scriptFile = reader.getAttributeValue(null,
"scriptFile");
        String style = reader.getAttributeValue(null, "style");
        String script = loadScript(scriptFile, resourceLoader);
        ComponentInfo componentType = loadComponentType(scriptFile,
resourceLoader);

        JavaScriptImplementation jsImpl =
factory.createJavaScriptImplementation();
        jsImpl.setComponentInfo(componentType);
        jsImpl.setScriptFile(scriptFile);
        jsImpl.setStyle(style);
        jsImpl.setScript(script);
        jsImpl.setResourceLoader(resourceLoader);
        return jsImpl;
    }
```

The load method is called positioned on the element to be handled (`<implementation.js>`) so that all attributes and content can be handled. The load method is responsible for parsing the XML stream and returning an appropriate `AssemblyObject` (in this case a JavaScriptImplementation). In this example, the loader is using the stream directly but it could just as easily use the data binding library of its choice. When the method returns, the reader should be positioned on the matching element end event.

In this case, the JavaScript implementation needs to access two external resources that are not part of the XML file. The source code for the script is loaded in the loadScript() method and stored as a property in the configuration model. Similarly the component definition is loaded from a `.componentType` sidefile by the loadComponentType() method and also stored in the model. The resulting `Implementation` object is thereby complete and the builder can create the final component without having to load external resources.

4.4 Contribute builders to construct your component context and invocation chains

a) Define your own component context

http://svn.apache.org/repos/asf/incubator/tuscany/java/sca/containers/container.rhino/src/main/java/org/apache/tuscany/container/rhino/context/JavaScriptComponentContext.java

```
public class JavaScriptComponentContext extends AbstractContext
implements AtomicContext {

    private Map<String, Class> services;

    private RhinoScript rhinoInvoker;

    private Map<String, Object> properties;

    private List<SourceWireFactory> sourceProxyFactories;

    private Map<String, TargetWireFactory> targetProxyFactories;

    private Object instance;

    public JavaScriptComponentContext(String name, Map<String, Class>
services, Map<String, Object> properties,
                                      List<SourceWireFactory>
sourceProxyFactories, Map<String, TargetWireFactory>
targetProxyFactories, RhinoScript invoker) {
        super(name);
        assert (services != null) : "No service interface mapping
specified";
        assert (properties != null) : "No properties specified";
        this.services = services;
        this.properties = properties;
        this.rhinoInvoker = invoker;
        this.sourceProxyFactories = sourceProxyFactories;
        this.targetProxyFactories = targetProxyFactories;
    }

...
}
```

b) Provide the ContextFactory to produce your context

http://svn.apache.org/repos/asf/incubator/tuscany/java/sca/containers/container.rhino/src/main/java/org/apache/tuscany/container/rhino/context/JavaScriptComponentContext.java

```
public class JavaScriptContextFactory implements
ContextFactory<AtomicContext>, ContextResolver {

    private Scope scope;

    private String name;

    private Map<String, TargetWireFactory> targetProxyFactories = new
HashMap<String, TargetWireFactory>();

    private List<SourceWireFactory> sourceProxyFactories = new
ArrayList<SourceWireFactory>();

    private Map<String, Class> services;

    private Map<String, Object> properties;

    private RhinoScript invoker;

    private CompositeContext parentContext;

    public JavaScriptContextFactory(String name, Scope scope,
Map<String, Class> services, Map<String, Object> properties,
RhinoScript invoker) {
        this.name = name;
        this.scope = scope;
        this.services = services;
        this.properties = properties;
        this.invoker = invoker;
    }

    public AtomicContext createContext() throws
ContextCreationException {
        return new JavaScriptComponentContext(name, services,
properties, sourceProxyFactories, targetProxyFactories,
invoker.copy());
    }

    public Scope getScope() {
        return scope;
    }

    public String getName() {
        return name;
    }

    public void addProperty(String propertyName, Object value) {
        properties.put(propertyName, value);
    }

    public void addTargetWireFactory(String serviceName,
TargetWireFactory factory) {
        targetProxyFactories.put(serviceName, factory);
    }
    public TargetWireFactory getTargetWireFactory(String serviceName)
{
        return targetProxyFactories.get(serviceName);
    }

    public Map<String, TargetWireFactory> getTargetWireFactories() {
        return targetProxyFactories;
    }

    public void addSourceWireFactory(String referenceName,
SourceWireFactory factory) {
        sourceProxyFactories.add(factory);
    }

    public void addSourceWireFactories(String referenceName, Class
referenceInterface, List<SourceWireFactory> factories, boolean
multiplicity) {
        sourceProxyFactories.addAll(factories);
    }

    public List<SourceWireFactory> getSourceWireFactories() {
        return sourceProxyFactories;
    }

    public void prepare(CompositeContext parent) {
        parentContext = parent;
    }

    public CompositeContext getCurrentContext() {
        return parentContext;
    }

}
```

c) Provide the ContextFactoryBuilder to create your ContextFactory

http://svn.apache.org/repos/asf/incubator/tuscany/java/sca/containers/container.rhino/src/main/java/org/apache/tuscany/container/rhino/builder/JavaScriptContextFactoryBuilder.java

```
public class JavaScriptContextFactoryBuilder extends
ContextFactoryBuilderSupport<JavaScriptImplementation> {

    @Override
    protected ContextFactory createContextFactory(String
componentName, JavaScriptImplementation jsImplementation, Scope
scope) {

        Map<String, Class> services = new HashMap<String, Class>();
        for (Service service :
jsImplementation.getComponentType().getServices()) {
            services.put(service.getName(),
service.getServiceContract().getInterface());
        }

        Map<String, Object> defaultProperties = new HashMap<String,
Object>();
        for (org.apache.tuscany.model.assembly.Property property :
jsImplementation.getComponentType().getProperties()) {
            defaultProperties.put(property.getName(),
property.getDefaultValue());
        }

        String script = jsImplementation.getScript();
        ClassLoader cl =
jsImplementation.getResourceLoader().getClassLoader();

        RhinoScript invoker;
        if (isE4XStyle(componentName,
jsImplementation.getComponentType().getServices())) {
            E4XDataBinding dataBinding =
createDataBinding(jsImplementation);
            invoker = new RhinoE4XScript(componentName, script,
defaultProperties, cl, dataBinding);
        } else {
            invoker = new RhinoScript(componentName, script,
defaultProperties, cl);
        }

        Map<String, Object> properties = new HashMap<String,
Object>();
        JavaScriptContextFactory contextFactory = new
JavaScriptContextFactory(componentName, scope, services, properties,
invoker);

        return contextFactory;
    }
```

d) Provide the WireBuilder to resolve and link the wires between the source and target

http://svn.apache.org/repos/asf/incubator/tuscany/java/sca/containers/container.rhino/src/main/java/org/apache/tuscany/container/rhino/builder/JavaScriptTargetWireBuilder.java

```
public class JavaScriptTargetWireBuilder extends
WireBuilderSupport<JavaScriptContextFactory> {

    protected TargetInvoker createInvoker(QualifiedName targetName,
Method operation, ScopeContext context, boolean downScope) {
        return new ComponentTargetInvoker(targetName, operation,
context);
    }
}
```

4.5 Register your extension as system components in the system module fragment

To add the loader/builder to the Tuscany runtime configuration, a `<component>` definition is added in a SCA Assembly file:

```
<moduleFragment xmlns="http://www.osoa.org/xmlns/sca/0.9"
xmlns:v="http://www.osoa.org/xmlns/sca/values/0.9"
xmlns:tuscany="http://org.apache.tuscany/xmlns/system/0.9"
name="org.apache.tuscany.container.rhino">

<!-- ContextFactoryBuilder -->
<component
name="org.apache.tuscany.container.rhino.builder.JavaScriptContextFac
toryBuilder">
  <tuscany:implementation.system
class="org.apache.tuscany.container.rhino.builder.JavaScriptContextFa
ctoryBuilder" />
  </component>

<!-- WireBuilder -->
<component
name="org.apache.tuscany.container.rhino.builder.JavaScriptTargetWire
Builder">
  <tuscany:implementation.system
class="org.apache.tuscany.container.rhino.builder.JavaScriptTargetWir
eBuilder" />
  </component>

<!-- ImplementationLoader -->
<component
name="org.apache.tuscany.container.rhino.loader.JavaScriptImplementat
ionLoader">
  <tuscany:implementation.system
class="org.apache.tuscany.container.rhino.loader.JavaScriptImplementa
tionLoader" />
</component>
</moduleFragment>
```

The extension is added to the system by including this fragment on the classpath as a resource with the name system.fragment.

Summary

In a nutshell, to extend Tuscany by contributing new implementation/binding types, you need to do the following things:

1. Define the SCDL extensibility element and corresponding model object
2. Contribute a StAX loader to load the XML configuration into the model object
3. Contribute a context factory builder to build runtime context based on the model
4. Contribute a wire builder to resolve and link wires for invocations

Disclaimer

This guide reflects the Tuscany M1 code level. The Tuscany extension model is subject to changes as the runtime evolves. Please check the Tuscany web site for the latest version.