SCA Java Extension Development Guide

Unable to render {include} The included page could not be found.

Unable to render {include} The included page could not be found.

Apache Tuscany SCA Java Extension Guide

This is a guide for developers who would like to extend Tuscany SCA Java.

- What is an extension?
- How to add a new component implementation?
- How to add a new binding?
- How to add a new interface binding?

- How to add a databinding
- How to add a new monitor extension?
- General guide for developing extensions
- Details on extension architecture **can move to architecture doc**
- Extension Modules

This page is under construction- You are welcome to help and complete it

A presentation on how to extend Tuscany in Microsoft PowerPoint format

Extending Tuscany PPT PDF

What is an extension?

Extension is a module (or a set of modules),when built with Tuscany SCA, can extend it to work with additional bindings, language implementations, programming models or databindings.

SCA assembly model is defined such that extensions can be added for new interface types or for new implementation types such as ruby and python, or new binding types such as CXF. However, the definition of extension SPIs is left to the implementor. Tuscany SCA Java provides a simple set of SPIs for implementing extensions. It also defines additional extension points such as data binding, for example support for SDO and JAXB.

This guide explains the steps to add each of these extension types. Please help us refine the guide as you go through the steps of adding extensions.

Name spaces used by extensions

Extensions sometimes contribute addtional elements or attributes to the SCA XML configuration model. These elements or attributes should be qualified as follows:

- http://www.osoa.org/xmlns/sca/1.0: if the extension is defined by the SCA spec.
- http://tuscany.apache.org/xmlns/sca/1.0: if the extesion is defined by Tuscany

How to add a new component implementation?

Musings on Adding a New Implementation Type by Mike E

How to add a new binding?

How to add a new interface binding?

How to add a new databinding?

How to add a new monitor extension?

Need for Monitor Extension

The need for monitor extension came as a requirement, to have a customized exception handling on Tuscany runtime especially when the user input errors are determined while reading, resolving and building phase of the contribution within the Tuscany runtime. Here different users/vendors using Tuscany with their products wanted to ignore some of the user input errors and allow the runtime to proceed further, while the others wanted to stop processing when a error is determined at the end of each phase. Another important need for a monitor extension is to validate the user inputs (via contributions) and report all the issues identified in one go (Example: As how the java compiler does for java code).

Understanding message conversion using monitor

1) Errors and warnings are logged with the monitor as problems and the processing continues. Exceptions won't be thrown, where ever possible. Processing here covers any processing of user input, i.e. contributions and the artifacts they contain.
2) After each phase namely read, resolve and build contribution, the problems are read from the monitor for display to the user (currently this will have already happened as our current monitor will just log errors and warnings as it receives them but you can imagine more capable monitors).
3) The controlling logic may look at the errors and warnings it is reporting to the user and prevent further processing occurring. Or it may allow processing to continue. This depends on which bit of processing we are talking about and what errors have been reported. Sample for the controlling logic is shown below

```
private void analyseProblems() throws Exception {
    for (Problem problem : monitor.getProblems()){
        // look for any reported errors. Schema errors are filtered
        // out as there are several that are generally reported at the
        // moment and we don't want to stop
        if ((problem.getSeverity() == Severity.ERROR) && (!problem.getMessageId().equals
("SchemaError"))) {
            if (problem.getCause() != null){
                throw new Exception(problem.getCause());
            } else {
                throw new Exception(problem.toString());
            }
        }
    }
}
```

Creating a new monitor extension

NOTE: Please have a look at the interface design of Monitor, Problem and MonitorFactory from monitor module in the Tuscany source code.

Step 1) Create a class MyMonitorImpl.java which implements the Monitor interface.

```
public class MyMonitorImpl implements Monitor {
    private static final Logger logger = Logger.getLogger(DefaultMonitorImpl.class.getName());
    // Cache all the problem reported to monitor for further analysis
    private List<Problem> problemCache = new ArrayList<Problem>();
    public void problem(Problem problem) {
        // Your custom code to handle the problem
    }
    // Returns a list of reported problems.
    public List<Problem> getProblems(){
        return problemCache;
    }
}
```

Step 2) Create a class MyMonitorFactoryImpl.java which implements the MonitorFactory interface.

```
public class MyMonitorFactoryImpl implements MonitorFactory {
    private Monitor monitor = null;
    public Monitor createMonitor() {
        if (monitor == null) {
            monitor = new MyMonitorImpl();
        }
        return monitor ;
    }
}
```

Step 3) Create a plain text file named as "META- INF/services/org.apache.tuscany.sca.monitor.MonitorFactory"
Register the extension by adding a line in this file as (org.apache.tuscany.sca.monitor.impl.MyMonitorFactoryImpl)

Initialize and create an instance of the monitor

Monitors can be initialized using the utility extensions as shown below.

```
UtilityExtensionPoint utilities = extensionPoints.getExtensionPoint(UtilityExtensionPoint.
class);
MonitorFactory monitorFactory = utilities.getUtility(MonitorFactory.class);
monitor = monitorFactory.createMonitor();
```

General guide for developing extensions

- Familiarize yourself with SCA 1.0 Assembly and the SCA Java 1.0 programming model. Specifications can be found at www.osoa.org.
- Never reference any classes in core. These classes are implementation-specific and subject to change; they are not part of the public SPI contract.
- Use autowire when assembling extension components.
- Do not play with classloaders such as setting the current context classloader unless it is absolutely necessary, i.e. a library used by an extension makes assumptions about context classloaders. Ideally the library can be refactored to not make these assumptions. If not, make sure the extension properly resets the current context classloader.

Detail on extension architecture

The ExtensionPointRegistry

```
package org.apache.tuscany.sca.core;


/**
* The registry for the Tuscany core extension points. As the point of contact
* for all extension artifacts this registry allows loaded extensions to find
* all other parts of the system and register themselves appropriately.
*
* @version $Rev: 539355 $ $Date: 2007-05-18 03:05:14 -0700 (Fri, 18 May 2007) $
*/
public interface ExtensionPointRegistry {

/**
* Add an extension point to the registry
* @param extensionPoint The instance of the extension point
*/
void addExtensionPoint(Object extensionPoint);

/**
* Get the extension point by the interface
* @param extensionPointType The lookup key (extension point interface)
* @return The instance of the extension point
*/
<T> T getExtensionPoint(Class<T> extensionPointType);

/**
* Remove an extension point
* @param extensionPoint The extension point to remove
*/
void removeExtensionPoint(Object extensionPoint);
}
```

The ModuleActivator

```
package org.apache.tuscany.sca.core;


/**
* ModuleActivator represents a module that plugs into the Tuscany system. Each
* module should provide an implementation of this interface and register the
* ModuleActivator implementation class by defining a file named
*
* "META-INF/services/org.apache.tuscany.core.ModuleActivator"
*
* The content of the file is the class name of the ModuleActivator implementation.
* The implementation class must have a no-arg constructor. The same instance
* will be used to invoke all the methods during different phases of the module
* activation. Note that the start and stop methods defined by this interface
* take a reference to the Tuscany SCA runtime ExtensionPointRegistry. This
* gives the ModuleActivator the oppotunity to add extension points to the
* registry as it is requested to start up and remove them when it is requested
* to shut down.
*
* @version $Rev: 564429 $ $Date: 2007-08-09 16:49:11 -0700 (Thu, 09 Aug 2007) $
*/
public interface ModuleActivator {

/**
* This method is invoked when the module is started by the Tuscany system.
* It can be used by this module to registr extensions against extension
* points.
*
* @param registry The extension point registry
*/
void start(ExtensionPointRegistry registry);

/**
* This method is invoked when the module is stopped by the Tuscany system.
* It can be used by this module to unregister extensions against the
* extension points.
*
* @param registry The extension point registry
*/
void stop(ExtensionPointRegistry registry);
}
```

The ModuleActivator represents a module that plugs into the Tuscany system. Each module can optionally provide an implementation of this interface and
register the implementation class by defining a file named as "META-INF/services/org.apache.tuscany.sca.core.ModuleActivator". The content of the file is the class name of the implementation. The implementation class must have a no-arg constructor. The same instance will be used to invoke all the methods during different phases of the module activation.

During bootstraping, the following sequence will happen:

1) All the module activators will be discovered by the presence of "META-INF/services/org.apache.tuscany.sca.core.ModuleActivator"
2) The activator class is instantiated using the no-arg constructor.
3) ModuleActivator.getExtensionPoints() is invoked for all modules and the extension points contributed by each module are added to the ExtensionRegistry.
4) ModuleActivator.start(ExtensionRegistry) is invoked for all the modules. The module can then get interested extension points and contribute
extensions to them. The contract bwteen the extension and extension point is private to the extension point. The extension point can follow similar
patterns such as Registry. If it happens that one extension point has a dependency on another extension point, they can linked at this phase.

During shutting down, the stop() method is invoked for all the modules to perform cleanups. A module can choose to unregister the extension from the
extension points.

Lazy instantiation of extensions

To reduce the startup time and memory footprint, most of the extension points in Tuscany now supports lazy instantiation of the registered extensions. The following describes the pattern using org.apache.tuscany.sca.contribution.processor.StAXArtifactProcessor as an example.

1) Create a plain text file named as "META-INF/services/org.apache.tuscany.sca.contribution.processor.StAXArtifactProcessor"
2) Register an extension by adding a line in the file

```
org.apache.tuscany.sca.implementation.java.xml.JavaImplementationProcessor;qname=http://www.osoa.org/xmlns/sca/1.0#implementation.java,model=org.apache.tuscany.sca.implementation.java.JavaImplementation
```

The simple syntax is: <implementation_class_name>;<parameter_name>=<parameter_value>,<parameter_name>=<parameter_value>