Extending an Application with Custom Plugins

In this tutorial, we will show how easily our application can be made extensible using the Struts 2 plugin mechanism. To keep the demonstration simple, our plugin will expose a JavaBean that writes a message. Plugins may include any combination of JavaBeans, Actions, Interceptors, Results or other resources we'd like available to an application.

The Interface

At runtime, plugins are retrieved and referenced via an Interface. So, first, we should define an interface that our plugin will implement. This interface must be available to both our web application and the plugin. To reduce coupling between the web application and the plugins, keep the interface in a separate JAR.

IMyPlugin.java

```
package example;

public interface IMyPlugIn {
   String saySomething();
}
```

The Plugin

Now that we have an interface to implement we'll create the plugin. At load time, the framework looks for JARs containing a `struts-plugin.xml` file at the root of the archive. To create a plugin, all we need to do is build a JAR and put the expected `struts-plugin.xml` at the root.

To get started, let's create a class that implements our IMyPlugin interface.

MyPlugin.java

```
package example.impl;

import example.IMyPlugin;

public class MyPlugin implements IMyPlugin {
   public String saySomething() {
       return "We don't need no education";
   }
}
```

Internally, the framework utilizes a number of JavaBeans. We can use the `bean` element to add our own JavaBeans to the set managed by the framework.

struts-default.xml

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>
    <bean type="example.IMyInterface" class="example.impl.MyPlugin" name="myPlugin"/>
</struts>
```

Finally, to install the plugin, drop the JAR file under WEB-INF/lib.

The Action

The JavaBeans configured by `bean` elements can be retrieved via a Container provided by XWork 2. We obtain a reference to the Container by using the `@Inject` notation. (This notation is part of the Guice framework that XWork and Struts use under the covers.) The framework predefines a Container object, and the `@Inject` annotation tells the framework to set its Container object to the Action property.

We might want to supply a number of JavaBeans to the application this way. In the Action, we will obtain a reference to the entire set of JavaBeans that might have been plugged in. Then, we can scroll through the set, displaying each JavaBean's message.

MyAction.java

```
package example.actions;

import example.IMyPlugin;

public class MyAction extends ActionSupport {
    Set<IMyPlugin> plugins;

    @Inject
    public void setContainer(Container container) {
        Set<String> names = container.getInstanceNames(IMyPlugin.class);
        plugins = new HashSet<IMyPlugin>();
        for (String name : names) {
            plugins.add(container.getInstance(IMyPlugin.class, name));
        }
    }

    public Set<IMyPlugin> getPlugins() {
        return this.plugins;
    }
}
```

As seen by the Action class code, it's important to define a unique interface for any beans that we plugin, so that we can identify our beans later.

In the same way that we plugged in this JavaBean, we could also plugin and configure Action classes, Interceptors, Results, or any other JAR-able resource that an application might utilize.

The JSP

Let's do something with those plugins:

Page.jsp

```
<s:iterator id="plugin" value="plugins">
  <s:property value="#plugin.saySomething()"/>
</s:iterator>
```