# How-To

## Table of contents

## 1. How to make your Eclipse plug-in aware of Spring beans

A plug-in context is provided to Eclipse plug-ins just like a Spring application context is provided to non-Eclipse applications by

1. adding the `info.smartit.eclipse.spring` plug-in to your Eclipse (or Eclipse RCP) installation. This plug-in requires the `org.springframework` plug-in and the `org.apache.commons.logging` plug-in. These required plug-ins must be added to your Eclipse (or Eclipse RCP) installation, too;
2. declaring a dependency on the `info.smartit.eclipse.spring` plug-in in your plug-in's `plugin.xml` file;
3. declaring an `info.smartit.eclipse.spring.pluginContext` extension; and
4. providing an XML file containing the bean definitions, which is named `pluginContext.xml` by default and located in the plug-in's root directory. This file must also be added to binary builds which is easily done by checking the appropriate box on the Build tab of your plug-in's manifest.

The term *your plug-in* references the plug-in which is to be developed.

The `info.smartit.eclipse.spring` plug-in behaves like an adapter to the Spring framework for Eclipse. It is based on some libraries from the Spring framework, which are provided unmodified through the `org.springframework` plug-in. As there are dependencies from the Spring framework to Apache's Jakarta Commons Logging (JCL), the JCL libraries are provided by the `org.apache.commons.logging` plug-in.

If your beans are defined in the default `pluginContext.xml` file, the plug-in context extension can simply be declared as follows: `<extension point="info.smartit.eclipse.spring.pluginContext"/>`.

The plug-in context file contains bean definitions in the very same format as `applicationContext.xml`, that is `<beans>`. Classpath resources which are referenced within the plug-in context are retrieved using your plug-in's classloader.

## 2. How to access beans in your Spring beans-aware Eclipse plug-in

There are (at least) three methods for accessing beans from your plug-in context. Which method to use depends on where you want to access the bean from. Of course you can use all these methods concurrently, even in the same plug-in.

### 2.1. Access bean from plug-in context via PluginContextBeanProxy

If you want to reference a bean in an extension which you declare in your `plugin.xml`

file, use `PluginContextBeanProxy` for the class attribute of your extension instead of the actual class, which is specified in the bean definition. You can think of that the proxy replaces itself by the bean from the default plug-in context which has the same name as the value of the `id` attribute in the tag which contains the class declaration.

The following example XML fragment from `plugin.xml` references a view bean named `my.view` which is assumed to be defined in the default plug-in context:

```
          <extension point="org.eclipse.ui.views">
            <view id="my.view"
class="info.smartit.eclipse.spring.PluginContextBeanProxy"/>
          </extension>
```

If the name of the bean differs from the value of the `id` attribute of the declaring XML tag, you can specify the beanname by appending a colon and the beanname to the proxy classname:

```
          <extension point="org.eclipse.ui.views">
            <view id="other.view.name"
class="info.smartit.eclipse.spring.PluginContextBeanProxy:my.view"/>
          </extension>
```

## 2.2. Access default plug-in context via Plugin instance

You can inject the default plug-in context into your plug-in by making your plug-in class implementation `ApplicationContextAware` and declaring your plug-in class as an abstract factory bean.

When using this pattern, your context-aware plug-in class should look something like this:

```
          package my.package;

          // imports omitted for brevity

          public class MyPlugin extends Plugin
          implements ApplicationContextAware {
              private static MyPlugin instance;
              private PluginContext pluginContext;

              private MyPlugin() {}

              public static MyPlugin getDefault() {
                  if (instance == null)
                      instance = new MyPlugin();
                  return instance;
              }
```

```
            // invoked on dependency injection processing by Spring;
            // your code never calls this method
            public void setApplicationContext(ApplicationContext context)
{
                // injected context is always PluginContext instance
                this.pluginContext = (PluginContext)context;
            }

            // optionally make the plug-in context accessible for
            // other, non-bean classes in this plug-in with code like:
            // pc = MyPlugin.getDefault().getPluginContext();
            public PluginContext getPluginContext() {
                return this.pluginContext;
            }

            // ...
        }
```

**Note:**

Your plug-in class needn't extend any special class for this method to work. The code above extends `Plugin`, but it may as well extend `AbstractUIPlugin` or any other class.

Your abstract factory bean definition in `pluginContext.xml` should read something like this:

```
        <bean class="my.package.MyPlugin" factory-method="getDefault"/>
```

## 2.3. Access any plug-in context via OSGi service

You can access any plug-in context - even a plug-in context which was declared by another plug-in - via the Plug-in Context Service.

**Unresolved Issue**

There is an unresolved issue when accessing plug-in contexts in other OSGi bundles which might not be active when the plug-in context is requested from the OSGi service.

In this case, your plug-in class should look something like this:

```
        package my.package;

        // imports omitted for brevity

        public class MyPlugin extends Plugin {
            private BundleContext bundleContext;

            public void start(BundleContext context) throws Exception {
```

```
                bundleContext = context;
            }

            // access any plug-in context by name
            public PluginContext getPluginContext(String name) {
                String sn = PluginContextService.class.getName();
                ServiceReference sr =
bundleContext.getServiceReference(sn);
                PluginContextService pcs = (PluginContextService)
                    bundleContext.getService(sr);
                PluginContext pc = pcs.getRequiredPluginContext(name);
                bundleContext.ungetService(sr);
                return pc;
            }

            // ...
        }
```

**Note:**

Your plug-in class needn't extend any special class for this method to work. The code above extends `Plugin`, but it may as well extend `AbstractUIPlugin` or any other class which implements the `IPlugin` interface.

The plug-in can provide its own default plug-in context by adding the following method to the above code:

```
            // variant for accessing this plug-in's default context
            public PluginContext getPluginContext() {
                String name = getBundle().getSymbolicName();
                return getPluginContext(name);
            }
```

However, the implementation for accessing the default plug-in context from the previous section is probably more efficient.