# Extensions-2-Services Manual

Neil Bartlett

October 22, 2009

## Contents

**1 Introduction** **1**
- 1.1 Overview of Extensions . . . 2
- 1.2 Overview of Services . . . 2
- 1.3 Challenges Using Services from Extensions . . . 2
  - 1.3.1 Lack of References to OSGi APIs . . . 2
  - 1.3.2 Lifecycle Mismatch . . . 3
  - 1.3.3 Factories Versus Singletons . . . 3
  - 1.3.4 No Explicit Release . . . 3

**2 Our Approach** **4**

**3 Development Guide** **5**
- 3.1 The Injected Factories Extension Point . . . 5
- 3.2 Using a Declared Factory from an Extension . . . 5
- 3.3 The Component Instance Class . . . 7
- 3.4 Reference Cardinality . . . 8
- 3.5 Default Bind Method Names . . . 8
- 3.6 Concurrency Issues . . . 9
- 3.7 The Service Lookup Strategy . . . 9

# 1 Introduction

Extensions-2-Services – or *e2s* as it will be referred to in this manual – is a framework for using Eclipse Extensions and OSGi Services in the same application. Extensions and services are both powerful but somewhat different approaches to the same problem, namely late binding in a modular application. Extensions tend to be used in Eclipse tooling plug-ins and Eclipse RCP (Rich Client Platform)

applications, whereas services tend to have more widespread use in many OSGi environments. Therefore when building Eclipse plug-ins and Eclipse RCP applications and using standard OSGi components, it is highly likely this will result in a mixture of extensions and services. These must be integrated, which is unfortunately non-trivial because of the differences in the way they work. *E2s* offers a solution that eases the integration task.

## 1.1 Overview of Extensions

The Extension Registry has been a fundamental part of Eclipse since its beginning. It defines the concept of *extensions* that are contributed to *extension points*. An *extension* is a declaration consisting of an XML sub-tree, which contains metadata about the functionality that is being contributed. For an example, the metadata for a "view" extension contains the name of the view, the icon to be displayed in the view's title bar, etc. It also contains the name of a class which implements the functionality of the view. An *extension point* is merely a declaration that a bundle expects to be extended, and it defines the content and format of metadata that extensions should offer.

Note that some extensions do not specify any class name, because they do not specify any functionality in terms of executable code. For example an extension may contribute "help" documentation to an application; this does not require programmatic code, only the location of the documentation and its title, language and so on. These kinds of purely declarative extensions are not interesting for the purposes of this manual, as we will shortly see.

## 1.2 Overview of Services

TODO

## 1.3 Challenges Using Services from Extensions

Let us suppose that we wish develop a view in Eclipse that shows the content of the OSGi log reader service, as defined in the Services Compendium. This requires us to access and call a service – `LogReaderService` – from an extension object implementing the `IViewPart` interface. There are several challenges we must overcome.

### 1.3.1 Lack of References to OSGi APIs

Extension objects such as views are instantiated by the Extension Registry using their default no-arg constructors. This means these objects are created with no

references to the `BundleContext` of their owning bundles and therefore cannot call the API required to lookup and obtain services.

### 1.3.2 Lifecycle Mismatch

Extensions and services exist on slightly different lifecycle. Services can only be registered or used by a bundle in the ACTIVE state, and this is the state that we have the most control over. Bundles move in and out of ACTIVE state whenever we wish them to, and therefore services can come and go many times during the lifetime of an application.

Extensions on the other hand are tied to the RESOLVED state of their owning bundle. We have very little control over RESOLVED state: though we can request a bundle to be resolved it may be implicitly resolved at other times; and there is no way to explicitly un-resolve a bundle except by completely removing it.

This makes it hard for extensions to *depend* on services. Between services, if service $A$ has a mandatory dependency on $B$ and $B$ goes away, then we simply make $A$ go away too. Extensions cannot be made to go away, so they cannot have mandatory dependencies.

### 1.3.3 Factories Versus Singletons

An extension represents a factory, therefore the client of an extension generally expects to receive a new, private object each time it calls `createExecutableExtension`. However, services are long-lived, shared objects – the same object is reused many times by many clients.

Some previous approaches to extensions/services integration have simply allowed for a service object to be supplied directly in response to the call to `createExecutableExtension`. This approach fails because some extension point client rely on factory behaviour. For example the `org.eclipse.ui.preferencePages` extension point is used to contribute pages to the main Preferences dialog, and each time the dialog is opened the workbench creates a new instance of the page object and discards it when the dialog is closed. Discarding causes the SWT resources inside it to be disposed, after which they are no longer usable. If we then force the workbench to reuse the same page object then it will attempt to reuse these previously disposed SWT resources, which results in `SWTError` exceptions.

### 1.3.4 No Explicit Release

When using services it is important to release a service when we have finished using it by calling `ungetService`. This allows OSGi to accurately track which bundles are using which services. If using the `ServiceTracker` API the `ungetService` call

is handled for us but the tracker itself consumes resources while it is active, and therefore it should also be released (by calling `close`) when we are done with it.

Unfortunately extensions provide no kind of release mechanism whereby the client of an extension object signals that it no longer requires the object. Returning to the preference page example, the workbench will generate new preference page objects each time the preferences dialog is opened, but there is no way for it to signal when those objects should be cleaned up – instead the references to them are released and they are allowed to be garbage collected.

A naïve approach to accessing services from an extension might be to create a `ServiceTracker` for each generated extension object. Unfortunately this will result in creating an ever-increasing number of trackers without ever closing them. While it is possible to hook into garbage collection events to close the tracker when the object is finally collected, this tends to be far too late.

# 2 Our Approach

*E2s* uses a conceptually simple solution to the above problems. It defines the idea of an "injected factory", which is a factory that manages references to one or more OSGi services. An injected factory is declared and configured via its own extension point; this is an approach that will be very familiar to Eclipse developers. Once configured, the factory is used as an alternative means of supplying new extension objects for any arbitrary extension point.

A factory may have zero or more references to services, and these references are managed at the factory level. Therefore if a bundle declares one factory containing one reference, then a maximum of one tracker is created. The factory uses the tracker to update all instances of the component class that it has previously created and have not yet been garbage-collected[1].

Much of the design of *e2s* is inspired by the Declarative Services specification from the OSGi Services Compendium. Unfortunately DS itself is not directly usable for our purposes but its style of referencing services and defining components is close to our needs.

[1]To allow the generated objects from a factory to be GC'd, the factory maintains only weak references to them. They therefore become eligible for collection as soon as they are no longer used by the client that requested their creation.

| Name | Type | Required? | Notes |
|---|---|---|---|
| `id` | String | Yes | The identity used to refer to the factory. |
| `class` | Java class name | Yes | The class which is to be instantiated by the factory. |

Table 1: Attributes of the `factory` element

# 3 Development Guide

## 3.1 The Injected Factories Extension Point

The `eu.wwuk.eclipse.extsvcs.core.injectedFactories` extension point allows us to define a factory where each instance object generated from the factory is "injected" with a service reference. Each instance is also subsequently notified when bound services become registered or unregistered. To declare a new injected factory, we create an extension to the `injectedFactories` extension point as follows:

```
<extension
      point="eu.wwuk.eclipse.extsvcs.core.injectedFactories">
   <factory
         id="logReaderView"
         class="org.example.view.LogReaderView">
      <reference
            cardinality="single"
            interface="org.osgi.service.log.LogReaderService">
      </reference>
   </factory>
</extension>
```

This declares a factory with the identity `logReaderView` that creates new instances of the `LogReaderView` class. The attributes of the `factory` element are shown in Table 1.

The `factory` element may contain zero or more `reference` elements which declare a binding to an OSGi service. In this case there is one reference to the `LogReaderService` defined by the OSGi Services Compendium specification, and that reference is singular, i.e. it binds only to one instance of the service. The attributes of the `reference` element are shown in Table 2.

## 3.2 Using a Declared Factory from an Extension

Whereas the `injectedFactory` extension point allows us only to *declare* a factory, we need a way to actually create and use the factory in an extension to an existing extension point. For example to create a View that may be opened by the user in the workbench, we have to declare an extension to the `org.eclipse.ui.views` extension point.

| Name | Type | Required? | Notes |
|---|---|---|---|
| `name` | String | No | The name of the reference, which can be used by instances to locate services. Defaults to the value of the `interface` attribute. |
| `interface` | Java class name | Yes | The interface name of the OSGi service to bind to. |
| `filter` | String | No | An optional additional filter which may be used to restrict the set of services that may be bound by the reference. |
| `cardinality` | [`single`\|`multiple`] | No | Whether to bind to a single instance of the service or all available. Defaults to `single`. |
| `bind` | String | No | The name of the method to call on the factory-generated objects when a service is bound. |
| `unbind` | String | No | The name of the method to call on the factory-generated objects when a service is unbound. |

Table 2: Attributes of the `reference` element

The connection between these is a special class `InjectionFactory` that we can pass, with some additional data, to the `class` attribute of *any* extension type. Therefore we can define our View extension as follows:

```
<extension
      point="org.eclipse.ui.views">
   <view
         id="org.example.views.logView"
         class="eu.wwuk.eclipse.extsvcs.core.InjectionFactory:logReaderView"
         name="Log Reader"
         icon="icons/log.gif">
   </view>
</extension>
```

This is exactly the same as a normal view declaration, except for the content of the `class` attribute. Note the use of the `InjectionFactory` class followed by a colon and then a factory ID – this is a hook back to the factory with the same ID that was declared in the previous section. Any extension point with a `class` attribute can use this approach.

## 3.3 The Component Instance Class

Listing 1 shows an example of a class that may be instantiated by an injected factory. Note that it is "POJO" inasmuch as it does not require any special interfaces to be implemented in order to benefit from the framework – the service instances are supplied and withdrawn via normal set and "unset" methods, also known as bind and unbind.

**Listing 1** An Example Component Class

```
public class LogReaderView extends ViewPart {
   @Override
   public void createPartControl(Composite parent) {
      Composite container = new Composite(parent, SWT.NONE);
      // ...
   }
   @Override
   public void setFocus() {
   }

   public void setLogReaderService(LogReaderService log) {
      // ...
   }

   public void unsetLogReaderService(LogReaderService log) {
      // ...
   }
}
```

## 3.4 Reference Cardinality

The cardinality of a service reference may be either single or multiple, where single is the default.

When single cardinality is used, the factory will call the bind method at most once to supply a single instance of the service. If more than one instance is available then the standard ranking approach – as documented in the JavaDocs for `ServiceTracker.getServiceReference()` – is used. If the currently-bound service instance becomes unavailable then the unbind method will be called and the reference will be in an unbound state. If a suitable replacement service is immediately available, or later becomes available, then the bind method will be called again to supply that instance. Note that the reference is never rebound as the result of a higher-ranked service than the currently bound one becoming available. The ranking of services is only considered when the reference is currently in the unbound state and there are many possible service instances that could be chosen.

When multiple cardinality is used, the bind method is simply called for every available service matching the filter, and the unbind method is called whenever one of those bound services becomes unavailable.

Note that both the single and multiple cardinalities permit there to be zero bound services at any time, i.e. they are equivalent to "`0..1`" and "`0..n`" cardinalities, and there is nothing equivalent to either "`1..1`" or "`1..n`". This is because extensions cannot be dynamically disabled, so we cannot make the extension available only when there are one or more service instances. Component implementation classes *must* be written to tolerate the absence of a service for periods of time, and this often has implications on the behaviour of the user interface. For example a Log Reader View should show an error placeholder when the `LogReaderService` is unavailable.

## 3.5 Default Bind Method Names

The methods `setLogReaderService` and `unsetLogReaderService` are used to *bind* and *unbind* a service to/from the component instance. The exact method names may be specified by supplying the `bind` and `unbind` attributes on the `reference` element.

However both `bind` and `unbind` attributes are optional, therefore if omitted, default method names are assumed based on the interface name of the OSGi service. First the package name of the interface is dropped, so `org.osgi.service.log.LogReaderService` becomes simply `LogReaderService`. Then a prefix is added, depending on whether the reference cardinality is single or multiple. If single, then the prefixes for bind and unbind are "set" and "unset" respectively.

If multiple cardinality then the prefixes are “add” and “remove”.

For example, a reference to the service `org.osgi.service.event.EventAdmin` with single cardinality will have default bind and unbind methods `setEventAdmin` and `unsetEventAdmin`. A multiple-cardinality reference to the service `org.foo.ConfigListener` will have bind and unbind methods `addConfigListener` and `removeConfigListener`.

## 3.6 Concurrency Issues

In OSGi, services can be registered or unregistered on any thread. Since the bind and unbind methods are called synchronously in response to service events, they can also be executed on any thread. Therefore it is imperative to use thread-safe coding practices when working with bound service objects.

It is also important to remember that GUI operations must only be performed on the main event thread. When implementing a bind/unbind method we cannot assume that the method will be executed on the GUI thread, so we should not perform GUI operations directly. Instead, pass a `Runnable` to the `asyncExec` method of the SWT Display object, as shown in Listing 2.

**Listing 2** Using `Display.asyncExec` to Perform GUI Operations

```
public void setLogReaderService(LogReaderService log) {
   final Enumeration entries = log.getLog();
   table.getDisplay().asyncExec(new Runnable() {
      public void run() {
         viewer.setInput(entries);
      }
   });
}
```

## 3.7 The Service Lookup Strategy

Since the dynamism of bind and unbind methods can complicate the programming of otherwise-simple objects, we offer an alternative mechanism for looking up service instances when the service is needed by the extension object.

The lookup method is named `locateService` and it exists on an interface named `ComponentContext`. We need a way to supply an instance of this `ComponentContext` to the extension object, therefore we define another interface `InjectedComponent` with one method, `setComponentContext`. When a factory creates a new object it will check if the object implements `InjectedComponent` and if so it will provide it with its context by calling `setComponentContext`.

The `locateService` method takes the name of one of the declared reference elements and returns a snapshot of the currently bound service instance, or `null`

if the reference is unbound. Note that the extension object must not hold onto the return value from `locateService` and reuse it over a period of time; it is intended to be used as quickly as possible and then discarded.