

Extensible Web Applications with Eclipse and Wicket

by Thomas Mäder

1. An IDE Gone Wild

Back around the turn of the millennium, IBM had quite a problem with its development tools. After the most reckless orgy of product renaming in modern history, the development tools were all named WebSphere-something-something, but that was, in fact, the only thing these programs had in common. IBM's idea of integration was opening the same file in the various editor programs.

So they turned to the lovely folks at OTI who had written VisualAge for Java and who had just recently completed VisualAge Micro Edition, their first file-based IDE. The mandate was to build a platform upon which the whole family of IBM development tools could be based. It quickly became obvious that no central agency could predict all the requirements the numerous IBM development tools would generate. Clearly the system had to be extensible, but even that was not enough. Even the plugin system had to be extensible. So they defined a mechanism to define extension points, and as a proof of concept, wrote the whole Eclipse platform using that mechanism. To that they added a class-loading and packaging mechanism which was eventually replaced by an OSGI implementation. Building systems out of Eclipse-style plugins has a couple of benefits. For one, it is easy to create product lines. If you want to create variations of a product, all you have to do is package the appropriate plugins and Bob's your uncle. Just look at the many packages you can download from the Eclipse site: you get downloads for plugin developers, C++ programmers, web developers, etc.

If you take this thought to the logical conclusion, you will end up with a system that is different for every single customer. Especially in markets where customers expect a large degree of configuration of a product for their particular needs (yes, I mean banking), you can tailor your product to the needs of every customer without polluting your base product. Another advantage is that third parties can extend your product in ways you never imagined. Just look at the literally thousands of Eclipse plugins you can download from the internet. You can publish a couple of extension points and (BAM!) you are now a platform, not a mere product.

But even if you don't build a product line or allow your customers to extend your product, structuring a program as extensions can improve maintainability through improved modularity and reduced coupling.

Many people writing plain old GUI applications saw the appeal of such a system (plus a ton of well tested, fully documented code to be reused) and soon enough non-IDE products were built on top of the refactored Eclipse RCP code base.

2. Adventures in the Barren Land

Back in 2003 I quit OTI/IBM and moved on to a company in the finance software business. They had built quite a software stack, including their own SOA framework and at the front end a Swing client. I'm not a fan of Swing for various reasons that have no bearing on this article. It felt like I had traded in a Harley Davidson for a bicycle: sure, you get places; it's just a such a pain sometimes. Little did I know that I was soon to travel on foot.

I must mention that the product we were building encountered quite some problems in the market place because of the Swing client. The (mostly small) banks we were trying to hock our wares to didn't like client software that had to be deployed to employees' work

stations. At some places, you were immediately out of any evaluation just on that account (unless, of course you were Microsoft or golfed with the CEO, but that's another story). At the same time, web applications had been somewhat rehabilitated by the arrival of „cool“ web applications, like Gmail, Flickr and the likes. Whatever the exact reasons, it was decided to port the Swing front end to the web.

Oh boy! What a complete fucking nightmare! It felt like crossing the ocean in a rowboat. Most so called web application frameworks would benefit from a trip to /dev/null. They make it nigh impossible to have a clean system design. Many times behaviour is defined in tag libraries or XML configuration files, thus making applications impossible to debug. There are a couple of frameworks that pretend to be based on components, but most do not offer any support for proper encapsulation of behaviour and (just as importantly) data. And, to be completely honest, I don't think much of the OO design of many web app frameworks. I guess the most brilliant programmers do not tend to gravitate towards Java web programming.

We finally decided to use JSF, and about 6 months into the experience, the whole team was just about ready to quit right after having ripped my head off (I had led the web framework evaluation, and honest, guv, on paper it all looked good). Enter Wicket. Too make a short story even shorter, we had rewritten the result of six months of work with JSF in three months (with a bunch of improvements) even though we had to learn Wicket first. So everyone was happy and leaving for the weekend on Wednesday because our work was done etc. etc. But we still had some problems left. Every bank we worked with had a different combination of features they wanted to buy. So certain pages and panels needed to be shown or hidden accordingly. Authentication and authorization mechanisms tended to be different. Some clients wanted extensions that made little sense for everyone else, or they wanted some workflow replaced. Sometimes errors occurred only at client sites, and some additional debug code was required. We started factoring out behaviour in classes that we loaded via `Class.forName()`. In short, we had many of the problems that the Eclipse plugin system solves.

We never seriously considered re-architecting our application as a plugin system, but I have since become convinced that modular web applications could be as much of a quantum leap as Eclipse was over previous development tools. There is actually a servlet bridge for eclipse that allows to start an OSGI instance inside a web application. However, I believe that Wicket is uniquely suited to build modular web applications because of it's true component nature. You could load extensions from plugin jars and because wicket components are true components (including data), it should just work.

3. The Basic Approach

The idea of a server side Eclipse is nothing new: there is, for example, the Eclipse servlet bridge (http://www.eclipse.org/equinox/server/http_in_container.php) or Pax Wicket (<http://wiki.ops4j.org/display/ops4j/Pax+Wicket>). What is common to those approaches is that they implement a particular application architecture. I believe that is not a good approach. Frameworks should grow out of concrete applications and therefore should not be technology-centric, but domain centric. What I mean by that is that if you build online banking applications, you will require a different application framework than if you are creating an online shop or mostly informational web site. And since framework architecture is largely expressed through extension points and extensions in an Eclipse application, It follows that the system I'm trying to build will not predefine any extension points.

One of the important architecture issues that arise is how much code to run inside or outside the OSGI container. An OSGI Container (like Eclipse's Equinox) creates a dedicated class loader for each OSGI bundle. Which classes are visible to which other classes is then controlled by declared dependencies between the bundles.

Java web applications, on the other hand, load their classes from files in the Web-Inf directory. The classes that form entry points to the application (i.e. Filters and Servlets) are loaded outside of the application's control. The point of entry for a Wicket application is always the WicketFilter class. The web application proper is then made up of a WebApplication instance, a session per user and a bunch of page objects.

In order to make this application extensible, we need to read extensions from the extension registry and maybe instantiate objects. Since we don't want to arbitrarily limit the places where we can have extension points, we want to run as much of the Wicket application inside the OSGI container as possible. What we want to use is similar to what the Eclipse servlet bridge does: we will create a web application that registers a HTTP filter with the servlet container (let's call it the „bridge filter“). At initialization time, the bridge filter starts the OSGI container. We must make sure that a particular bundle (the „extension bundle“) is started immediately and this extension bundle will create a wicket filter and register it with the bridge filter. The bridge filter then forwards every request to the Wicket filter inside the OSGI container.

4. Creating the Web Application

If you have any experience with Wicket, you'll know how to create and run a web application, so I'll leave that part as an exercise for the reader. Our System will consist of two cooperating parts: the bridge filter and the extension bundle. The extension bundle contains the wicket libraries and it is responsible for loading the WicketFilter and registering it with the servlet filter. Let's look at the init() method in our BridgeFilter class:

```
public void init(FilterConfig config) throws ServletException {
    filterConfig= config;
    framework = new FrameworkLauncher();
    framework.start(config.getServletContext());
}
```

It remembers the filter configuration and starts up the OSGI framework (we'll get to that in a second). The OSGI container will start up our bundle (ch.devotek.stump.extensionbundle) and call the bundle's Activator.start() method.

```
public void start(BundleContext context) throws Exception {
    bundleContext= context;
    BridgeFilter.registerFilter(new WicketFilter());
}
```

We just instantiate a WicketFilter (which is on the local bundle class path) and register it with our BridgeFilter. We need to have the wicket libraries (wicket.jar, etc.) on the bundle class path in order to instantiate the filter, of course. Here's the BridgeFilter.registerFilter() method:

```
public static synchronized void registerFilter(Filter filter) throws
ServletException {
    filter.init(filterConfig);
    delegate = filter;
}
```

We just remember the registered filter in a static variable. Now that the BridgeFilter knows the WicketFilter instance from inside the OSGI Container, it can forward every call to

BridgeFilter.doFilter(...) to the WicketFilter instance. Note that we initialize the WicketFilter with the filter configuration we got for the bridge filter. This allows us to specify the name of the Wicket application class just as we would for a regular Wicket application like so:

```
<filter>
  <filter-name>eclipseFilter</filter-name>
  <filter-class>ch.devotek.stump.bridge.BridgeFilter</filter-class>
  <init-param>
    <param-name>applicationClassName</param-name>
    <param-value>ch.devotek.stump.app.example.ExampleApplication</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>eclipseFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

What's missing from the picture now is how to start the OSGI container. Fortunately, there is a class called EclipseStarter that comes with the Eclipse distribution. We'll need to copy the file org.eclipse.osgi_<version string>.jar to our web application's Web-Inf/lib directory (and add it to the compile class path, of course). The code to start the OSGI container is in the class FrameworkLauncher:

```
public synchronized void start(ServletContext context) {
    Map<String, String> initialPropertyMap =
buildInitialPropertyMap(context);
    try {
        EclipseStarter.setInitialProperties(initialPropertyMap);
        EclipseStarter.startup(new String[] { "-consoleLog" }, null);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

That's simple, eh? Of course, the interesting part is in buildInitialPropertyMap()...

```
private Map<String, String> buildInitialPropertyMap(ServletContext context) {
    Map<String, String> initialPropertyMap = new HashMap<String, String>();

    File tmpDir=(File) context.getAttribute("javax.servlet.context.tempdir");

    try {
        initialPropertyMap.put(OSGI_CLEAN, "true");
    }
```

This just means that we always clear out the OSGI bundle cache. In production, this would not be necessary

```
File configurationDirectory = new File(tmpDir, "configuration");
if (!configurationDirectory.exists()) {
    configurationDirectory.mkdirs();
}
```

```

    }
    initialPropertyMap.put(OSGI_CONFIGURATION_AREA,
        configurationDirectory.toURL().toExternalForm());

```

OSGI uses this directory to put configuration data. We'll just use a new directory in the web application's temp directory

```

    initialPropertyMap.put(OSGI_INSTANCE_AREA,
        new File(tmpDir, "workspace").toURL().toExternalForm());

```

The instance data area. In the Eclipse IDE, this would be the workspace.

```

    initialPropertyMap.put(OSGI_BUNDLES, buildBundleList());

```

Here, we set the list of bundles to install and start.

```

    initialPropertyMap.put(OSGI_DEV, "bin");

```

This adds the „bin“ folder to the runtime classpath of every bundle. We need this in order to run directly from the Eclipse workspace.

```

    initialPropertyMap.put("osgi.parentClassLoader", "app");

```

Now this is important: It sets the web application class loader as the parent of the OSGI class loaders. We need this so that our web application can see the classes from the servlet API.

The rest is just error handling. The last piece of the puzzle that's missing is the list of bundles to be installed and started. The method `buildBundleList()` builds a comma separated list of bundles of the form

```
file://<path to workspace project or jar>[@start]
```

In addition to the OSGI System bundle (which we get for free), we need the bundles „org.eclipse.equinox.common“ and „org.eclipse.equinox.registry“ in order to run the eclipse plugin registry. We also need to start „ch.devotek.stump.extensionbundle“ in order for our bundle activator to run and register the `WicketFilter` with the `BridgeFilter` class.

You can try and run this, but you will get a `ClassNotFoundException` because we have specified the class „ch.devotek.stump.app.example.ExampleApplication“ as the Wicket application to run in our web.xml file, but we haven't actually implemented that class.

5. An Example Application

As an example, we'll define an extension point for main pages. The main pages will extend the `MainPage` class which populates the navigation bar from an extension point.

So we create a plugin project called „ch.devotek.stump.app.example“ and define an extension point „mainPages“. The entry in the plugin.xml must have two attribute: „class“ and „label“. We can then build our main navigation like so:

```
final ListView navigation= new ListView("navigation", new
LoadableDetachableModel() {
```

```

    @Override
    protected Object load() {

```

```

        IExtensionRegistry registry = RegistryFactory.getRegistry();
        final IConfigurationElement[] configs = registry
            .getConfigurationElementsFor("ch.devotek.stump.app.example",
                "mainPages");
        return Arrays.asList(configs);
    }
}
}) {
    @Override
    protected void populateItem(ListItem item) {
        IConfigurationElement config= (IConfigurationElement)
            item.getModelObject();
        String label = config.getAttribute("label");
        final String clazz= config.getAttribute("class");

        Link link = new Link("link") {
            @Override
            public void onClick() {
                setResponsePage(Classes.resolveClass(clazz));
            }
        };
        item.add(link);
        link.add(new Label("label", label));
    }
};

```

Note the use of a `LoadableDetachableModel` because configuration elements are not serializable. We can then add a the home page in our `plugin.xml` (and add the corresponding `MainPage` subclass, of course).

```

<extension
    point="ch.devotek.stump.app.example.mainPages">
    <mainPage
        class="ch.devotek.stump.app.example.HomePage"
        label="Label">
    </mainPage>
</extension>

```

All we need to do is create the class `ExampleApplication` we mentioned in the `web.xml` and all should be well. Wrong! Wicket will not be able to load our application class, because it is not visible from the extension bundle, where the the code to create the application instance is run. Obviously, we cannot make the extension bundle depend on the example application bundle; that would defeat the point of making an extensible application. The trick is to add the line

Eclipse-BuddyPolicy: dependent

to the `MANIFEST.MF` file of the extension bundle. This allows the extension bundle to load classes from all bundles that require the extension bundle (directly or indirectly). But that is not sufficient: Wicket uses an implementation of `IClassResolver` to look up classes from class names. This is used, for example, to resolve the classes of bookmarkable pages.

The default implementation uses the context class loader, which will not work when called from the web server. What we want to do is use the extension bundle to resolve classes. So our resolver will just look like this:

```
public Class<?> resolveClass(String classname) throws ClassNotFoundException {
    BundleContext bundleContext = Activator.getBundleContext();
    return bundleContext.getBundle().loadClass(classname);
}
```

Now we can go and write more main pages and they will automatically appear in the main navigation. I've written a page that shows the current time via AJAX in a plugin „ch.devotek.stump.app.example.timepage“.

Still Missing

There are a couple of things that would have to be sorted out in order to get from the spike solution I've described above to a proper production system.

The first is certainly the packaging. If I've understood correctly, Equinox cannot run out of a war file. We'd have to devise a way to either preinstall the plugins somewhere or unpack them to a writable location (as the Eclipse servlet bridge does).

Eclipse offers a couple of extensions to the OSGI model that would be interesting to pursue: particularly, I'm thinking of product builds, auto-update and automatic discovery of installed plugins.

But most important: it would be really interesting to actually build a real Wicket application with Eclipse. Only real world experience can show whether it's worth the hassle in the end.