Instructions for the enRoute Blog Tutorial

Peter Kriens

Table Of Content

Disclaimer

This document does not contain any official OSGi material. Any information here is subject to change and has not been agreed to by the OSGi Alliance board. Much of it is personal opinion of an opinionated person.

1. enRoute Blog Tutorial

Prerequisites

1. Ensure that you have Java 7 installed on your system. That is, when you go to the command line (yes, this tutorial will include command lines). If you do not have Java installed then this tutorial might be too advanced. If you want to try anyway, download the latest Java 7 for your platform from: http://www.java.com/en/download/index.jsp
2. Have a valid and recent Eclipse installation. We have tested with Eclipse Kepler. You can download Eclipse from http://www.eclipse.org/downloads/. Suggested is to use the 'Eclipse IDE for Java Developers', this is the smallest version and can easily be extended through the market place.
3. Ensure you have set in Eclipse's preferences that all files are UTF-8. Really.
4. From the market place, install bndtools. bndtools is a plugin that provides extensive support for OSGi development. Alternatively you can install the latest and the greatest from https://bndtools.ci.cloudbees.com/job/bndtools.master/lastSuccessfulBuild/artifact/build/generated/p2/
5. Make sure you have the following plugins:
   - WTP HTML / JS Editor
   - Markdown Editor
   - Console Plugin (Google)
6. Ensure git is installed: http://git-scm.com/book/en/Getting-Started-Installing-Git
7. Install jpm: http://jpm4j.org/#!/md/install
8. Install bnd with jpm: `sudo jpm install bnd@*`
9. Clone the git workspace from Github

   ```
   # where you want your workspace to appear,
   # make sure not spaces in paths
   cd ...
   git clone git clone git@github.com:osgi/osgi.enroute.blog.git
   cd osgi.enroute.blog
   ```

   This will create a workspace, called `osgi.enroute.blog`. In the remainder of this document this is referred to as the *workspace*. In the accompanying image it is in `~/workspace/osgi.enroute.blog`

Legend

- `code` — Text in the code font refers to names that are used as links. Variable names, file names, bundle names, etc.
- `menu` — Any references to the graphic user interface are in this font.
- *term* — Term definition. Is usually followed with an explanation.
- **`changed/added`** — Text in this font must be added to a source file (bnd.bnd, javascript, html, etc).
-
- - make sure to clean the workspace,

2. Setup: Running a Framework

Goal

To get a framework running, inspecting what tools are available for us to develop and thus debug applications, and provide configuration data from a project.

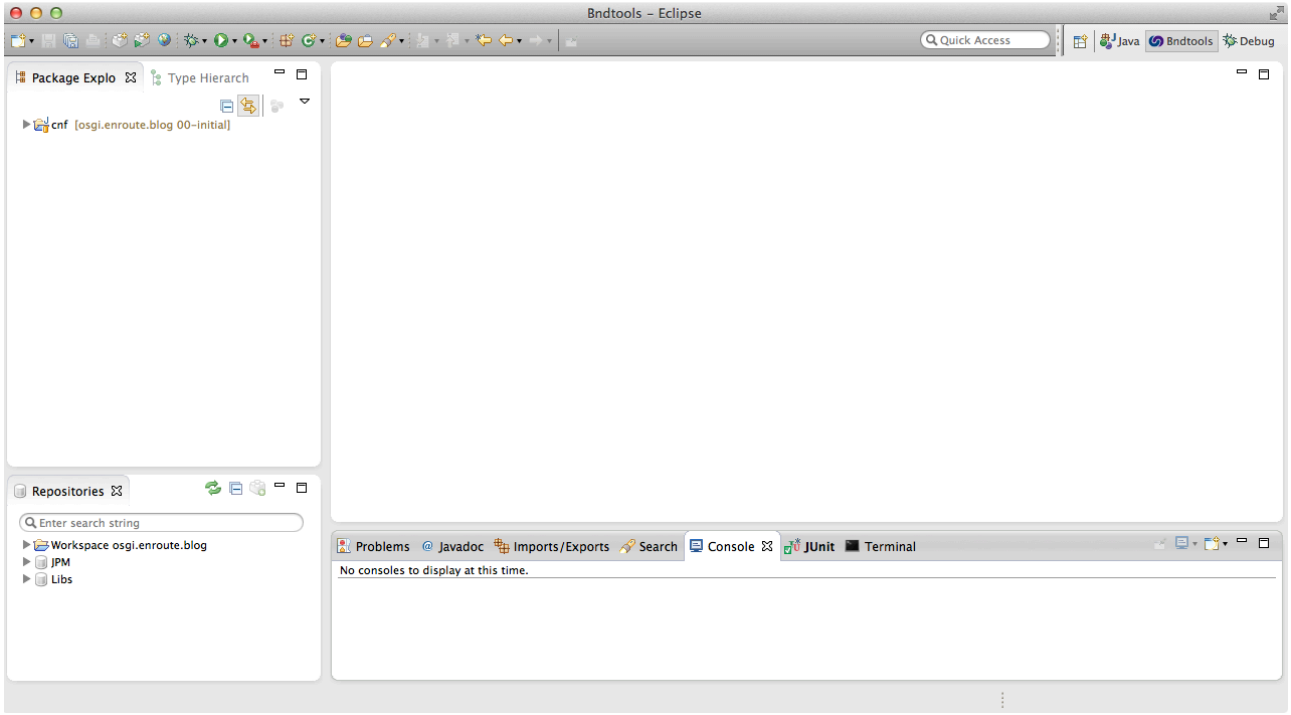
Prerequisites

First ensure that you have checked out the initial branch in the workspace.

```
$ cd osgi.enroute.blog
$ git checkout 00-initial
```

It is advised not to store Eclipse metadata in the git workspace. This allows you to clean the actual workspace at will, So create a directory `metadata` and in there create a workspace `osgi.enroute.blog`. Once this workspace is started, import all projects from `workspace/osgi.enroute.blog`. It is crucial that any project is created in this `workspace/osgi.enroute.blog` directory; a bnd project must have the `cnf` directory relatively reachable as `../cnf`.

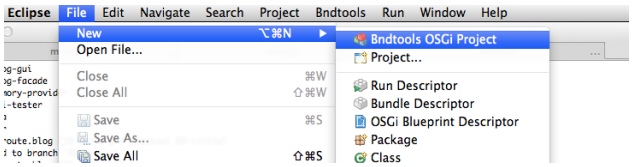
You should have bndtools open on the workspace. It should look similar to the following workspace.



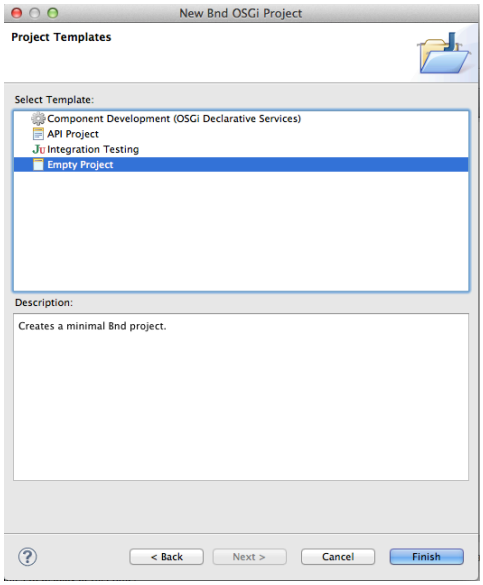
The only project is `cnf`, this directory defines the setup of the workspace and provides shared information like build files, etc.

Creating the Application

In bndtools you can create a new Application from the File menu:



You should name the project `osgi.enroute.blog.appl`. For this tutorial we do not use a prepared template so choose an *empty project*.



Setting up

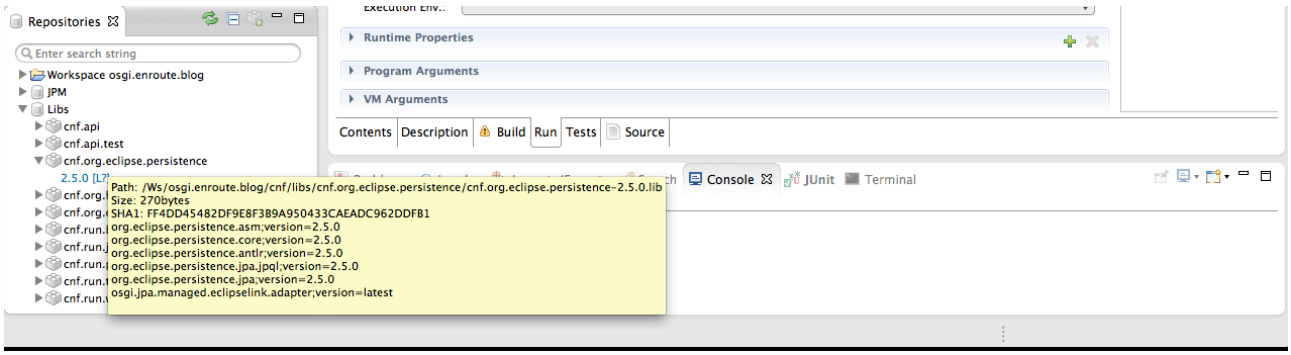
To setup the project, go into the project and double click the `bnd.bnd` file. After creating a new project it is best to fill in the version right away. The best version is:

```
1.0.0.${tstamp}
```

The `${tstamp}` is a macro that expands to the current date/time.

It also helps to fill in the description, this makes it easier for others to understand what the bundle will do.

Now select the `Run` tab and a number of prepared configurations. These configurations can be found in the `Repository` view, in the Libs repository. You can open the Repository view from the `View/Other/Repository` menu.



The `Libs` repository contains entries, so called *libraries*, that refer to a number of bundles. You can hover over an entry to see where it is stored as well as its contents.You can edit the contents in the `cnf` project, in the `libs` directory.

Drag the following libraries to the `Run Bundles` pane (on the right).

- `cnf.run.base` - Provides basic bundles.
- `cnf.run.web` - Provides the basic Jetty web server and whiteboard support from Apache Felix.
- `cnf.run.web.debug` - Provides Web Console, shell, and XRay

The bndtools user interface is a shell over the `bnd.bnd` text file. Double click on the `bnd.bnd` file and then select the `Source` tab to see what we have so far :

```
Bundle-Version:                 1.0.0.${tstamp}
Bundle-Description:     \
 A simple Blog Application root bundle. The application runs\
 as a web server and provides a GUI to list, create, \
 update, and delete blogs.

-runbundles: \
    cnf.run.base, \
    cnf.run.web, \
    cnf.run.web.debug
```

Feel free to edit the source file, changes can be made in the text as well as in the graphic user interface (GUI). When editing the `Source` tab, make sure that there is no space between the \ and the newline. The standard bnd editor colors this red but it is easy to miss.

We need to make one more change, otherwise bnd will raise a warning if we try to run this project now since there is no content yet. So create a `readme.md` file with a text editor to describe the bundle and include it in the bundle:

```
-includeresource: \
  {readme.md}
```

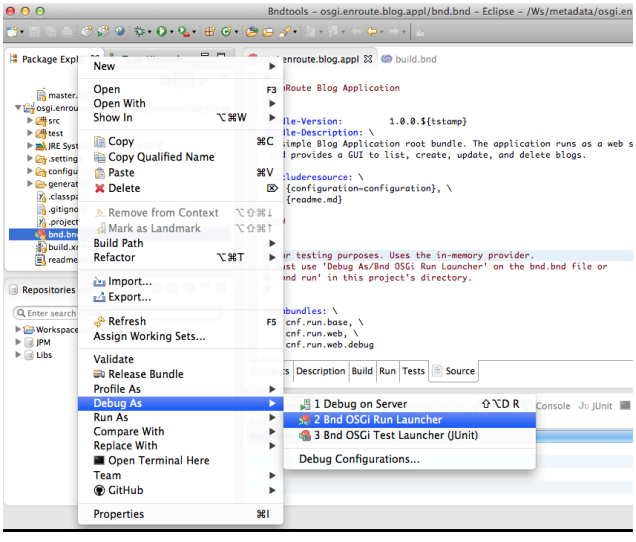
The curly braces around readme.md tell bnd to preprocess the text file, this means you can use any macro. In general, you like to run text files used in the build through the pre-processor.

This bnd.bnd file inherits information from the workspace, see cnf/build.bnd. Among some standard headers it also inherits the framework and some other settings:

```
-runfw: org.apache.felix.framework;version='[4,5)'
-runee: JavaSE-1.7
-runvm:
```

Running

This together is sufficient to run this, admittedly rather useless, bundle. However, it will show us the setup of the framework. Select the bnd.bnd file and call up the context menu. Then select `Debug As/Bnd OSGi Run Launcher`.



Gogo Shell

This will give you a shell in the Eclipse Console. You can stop the running framework by clicking the red square.

You can now see all the installed bundles by typing `bundles`.

```
g! bundles
    0|Active      |    0|org.apache.felix.framework (4.2.1)
    1|Active      |    1|org.apache.felix.configadmin (1.6.0)
    2|Active      |    1|org.apache.felix.log (1.0.1)
    3|Active      |    1|org.apache.felix.scr (1.6.2)
    4|Resolved    |    1|slf4j.simple (1.7.5)
    5|Active      |    1|slf4j.api (1.7.5)
    6|Active      |    1|aQute.configurer (1.0.1.201310081544)
    7|Active      |    1|aQute.executor (1.0.0.201306251215)
    8|Active      |    1|org.apache.felix.gogo.runtime (0.10.0)
    9|Active      |    1|org.apache.felix.gogo.shell (0.10.0)
   10|Active      |    1|org.apache.felix.gogo.command (0.12.0)
   11|Active      |    1|aQute.logger.intrf (1.0.0.201308271446)
   12|Active      |    1|org.apache.felix.http.jetty (2.2.0)
   13|Active      |    1|org.apache.felix.http.whiteboard (2.2.0)
   14|Active      |    1|aQute.webserver (1.0.6.201309091018)
   15|Active      |    1|aQute.rest.srv (2.0.0.201310211450)
   16|Active      |    1|aQute.services.struct (1.0.0)
   17|Active      |    1|org.apache.felix.metatype (1.0.8)
   18|Active      |    1|org.apache.felix.Web Console (3.1.6)
   19|Active      |    1|aQute.xray.plugin (1.1.0.201310101226)
   20|Active      |    1|osgi.enroute.blog.appl (1.0.0.201310230917)
```

You can type `help` to see the commands in the shell. The most common commands are:

- `help` — A list with all commands
- `help <scope:command>` — Help for a specific command, e.g. `help scr:list`
- `bundles` — List all bundles (this is actually directly calling `BundleContext.getBundles`).
- `start <n>` — Start bundle n (also `(bundle <n>) start`)
- `stop <n>` — Stop bundle n (also `(bundle <n>) stop`)
- `bundle <n>`

This gogo shell is very powerful, it supports piping, variables, closures, and much more. You can read more about this shell at http://felix.apache.org/site/rfc-147-overview.html

Web Console

Though shells are incredibly important, it is often easier to use web based tools since this can convey more information in a smaller space. Among the installed bundles there is a web server and the Apache Felix Web Console. The webserver is started by default at port 8080, the Web Console registers on /system/console. So you can go to your browser and select http://localhost:8080/system/console. The default login is `admin/admin`.

You will be redirected to the `Bundles` pane, listing all bundles.

Apache Felix Web Console
Bundles

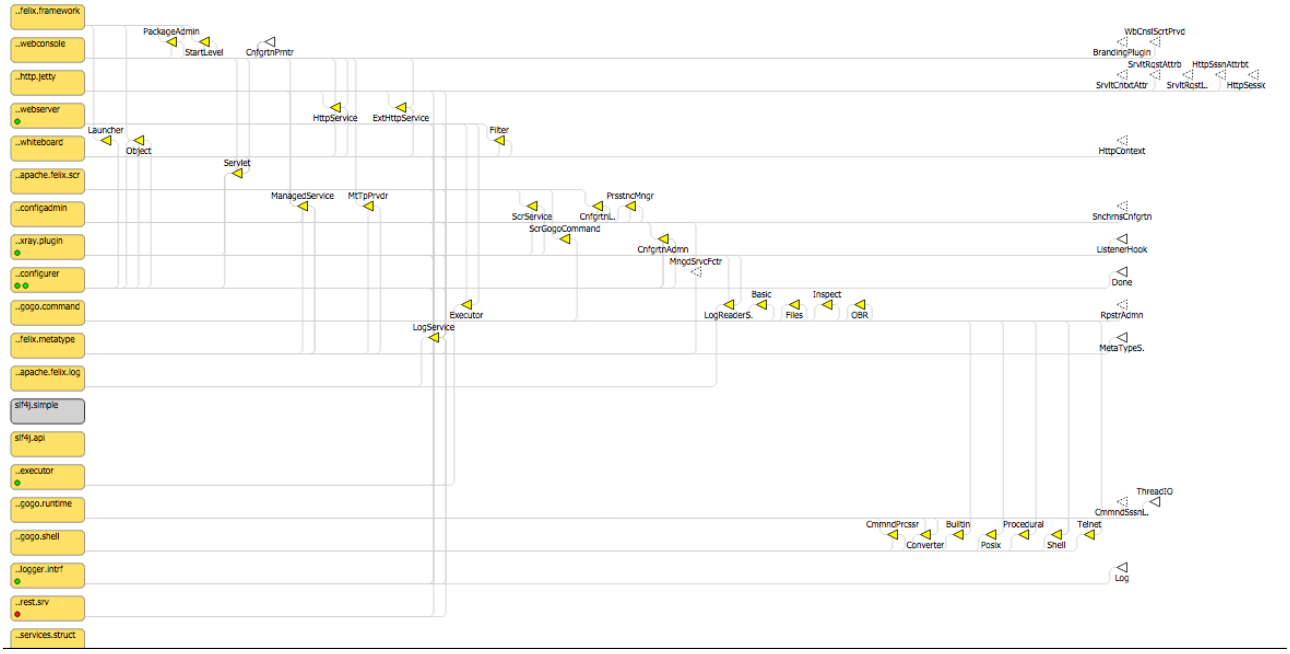
Bundles | Components | Configuration | Configuration Status | Licenses | Log Service | OSGi Repository | Services | Shell | System Information | X-Ray

Bundle information: 21 bundles in total, 20 bundles active, 1 active fragments, 0 bundles resolved, 0 bundles installed.

Apply Filter | Filter All | Reload | Install/Update... | Refresh Packages

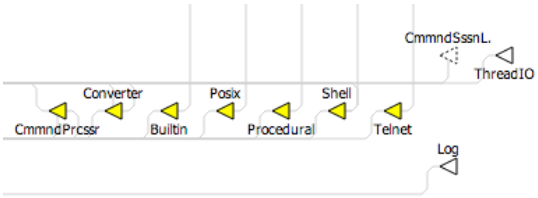
Id	Name	Version	Category	Status	Actions
0	System Bundle *(org.apache.felix.framework)*	4.2.1		Active	
1	Apache Felix Configuration Admin Service *(org.apache.felix.configadmin)*	1.6.0	osgi	Active	
3	Apache Felix Declarative Services *(org.apache.felix.scr)*	1.6.2	osgi	Active	
10	Apache Felix Gogo Command *(org.apache.felix.gogo.command)*	0.12.0		Active	
8	Apache Felix Gogo Runtime *(org.apache.felix.gogo.runtime)*	0.10.0		Active	
9	Apache Felix Gogo Shell *(org.apache.felix.gogo.shell)*	0.10.0		Active	
12	Apache Felix Http Jetty *(org.apache.felix.http.jetty)*	2.2.0		Active	
13	Apache Felix Http Whiteboard *(org.apache.felix.http.whiteboard)*	2.2.0		Active	
2	Apache Felix Log Service *(org.apache.felix.log)*	1.0.1		Active	
17	Apache Felix Metatype Service *(org.apache.felix.metatype)*	1.0.8	osgi	Active	
18	Apache Felix Web Management Console *(org.apache.felix.webconsole)*	3.1.6		Active	
6	aQute.configurer *(aQute.configurer)*	1.0.1.201310081544		Active	

While developing bundles, you will visit a lot of these panes. However, there is one pane that provides a graphic overview about the health of your system. This is *X-Ray*. X-Ray shows the bundles vertically, with their DS components as green or red LEDs, and the services horizontally. By hovering over a service or bundle you can see the incoming and outgoing connections. That is, other connections are temporarily not displayed. Go to http://localhost:8080/system/console/xray to see X-Ray.

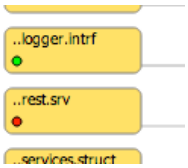


Bundles display their state through color coding. Orange is `ACTIVE`, red is `STARTING`, white is `INSTALLED`, and `RESOLVED` is gray. You can click on the bundle's icon to go to the Bundle's tab in the Web Console at the given bundle. Hovering over the bundle icon shows you the exact name.

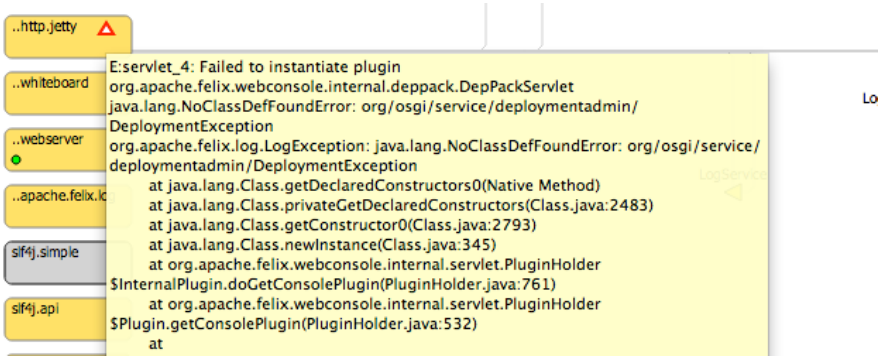
Services are yellow if they are in use. A white service with a black outline registered but not used, when the outline is dashed there is a bundle looking for it. The white services are displayed at the right. If something is not working, look at these services because they often indicate that bundles/components are looking for services are or are not used at all. Services can be clicked and this will make the page go to the Services tab. However, the exact service is not selected since each service icon can represent multiple service objects. That is, multiple bundles can register the same service name; the display only shows one icon per name and there can thus be multiple bundles registering that service.



DS Components are show with a green or red status LED. Green indicates that the component is active, red indicates that the component is not satisfied; this sometimes means there is no configuration record available. You can click on the LED to go to the configuration tab.



As said, things fail. These message are normally stored in the log. However, one often forgets to look in this log. If a bundle records a log ERROR event then a warning icon is shown on that bundle. Hovering over that icon will show you that log record, including a stack trace. This icon stays up until two minutes after the error occurred.



X-Ray is full dynamic, it will continuously poll the framework and update its status. Even if the framework has been restarted, X-Ray will pick up the state again if the framework becomes active again.

Configuration

The framework runs but is not configured at all. We therefore need to configure it. In the current setup there is a bundle that allows you to provide configuration information for Configuration Admin: `aQute.configurer`. This bundle will read configuration in JSON format from the bundle's `configuration/configuration.json` file. We therefore need to add such a file.

The contents for an initial file can look like:

```
[
    {
        "service.pid" : "org.apache.felix.http",
        "org.apache.felix.http.enable" : true,
        "org.osgi.service.http.port" : 8080,
        "org.apache.felix.http.debug" : true,
        "org.apache.felix.https.enable" : false
    }, {
        "service.factoryPid" : "aQute.webserver.WebServer",
        "service.pid" : "WebServer",
        "exceptions" : true,
        "alias" : "/"
    }, {
        "service.factoryPid" : "aQute.executor.ExecutorImpl",
        "type" : "FIXED",
        "service.pid" : "DefaultExecutor",
        "size" : 16
    }
]
```

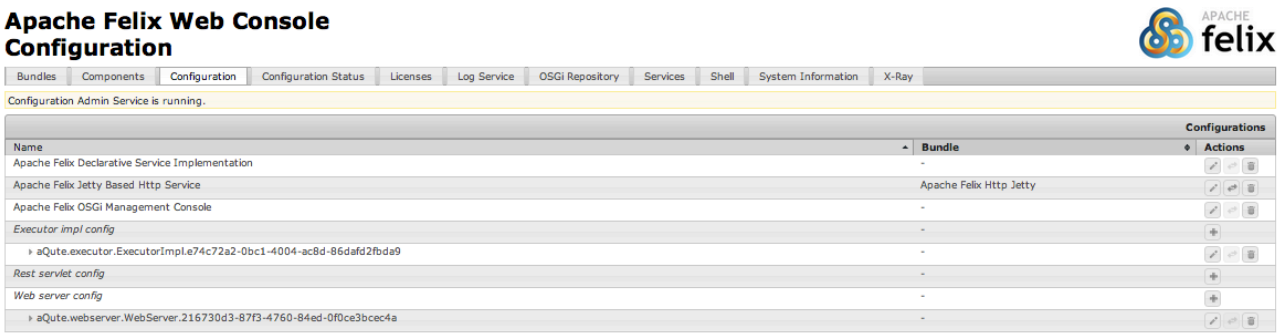
This provides configurations for the Apache Felix http server, the Webserver (provides extender support for bundles delivering content) and an Executor shared by all bundles.

This file is not included by default, we need to add another clause to the `bnd.bnd` file's `-includeresource` instruction to include it. This instruction will then look like:

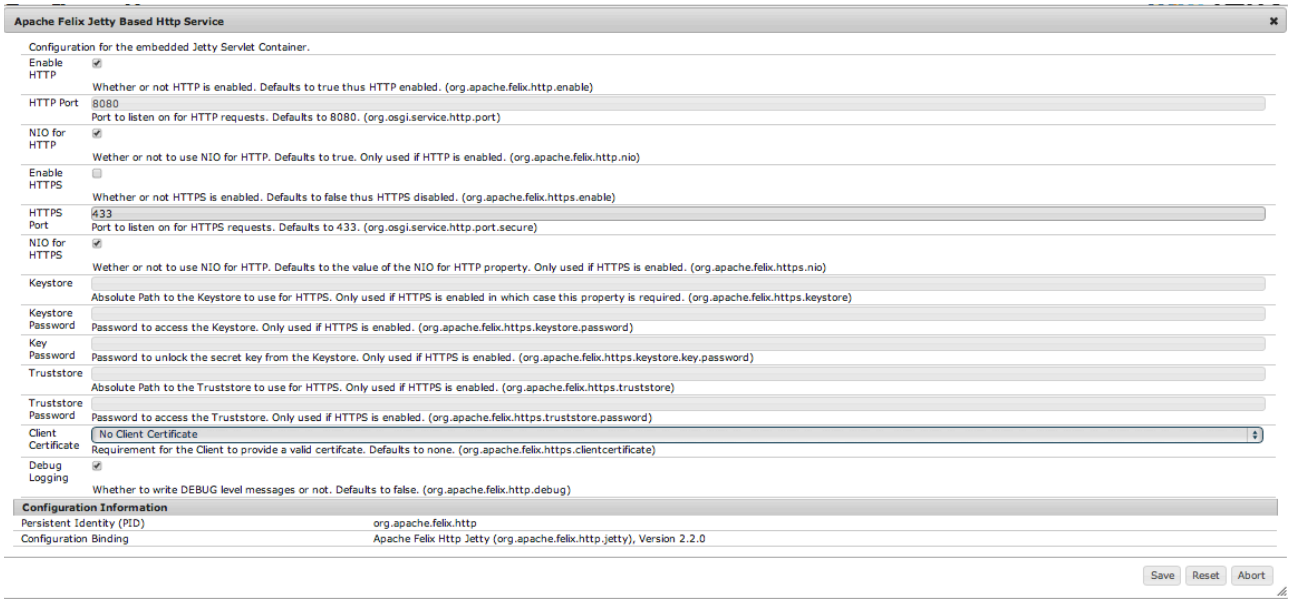
```
-includeresource: \
    {configuration=configuration}, \
    {readme.md}
```

As you can see now, we also pre-process the configuration file. The moment you save this file, bnd will generate the bundle and will update this bundle in the running framework. This will happen silently.

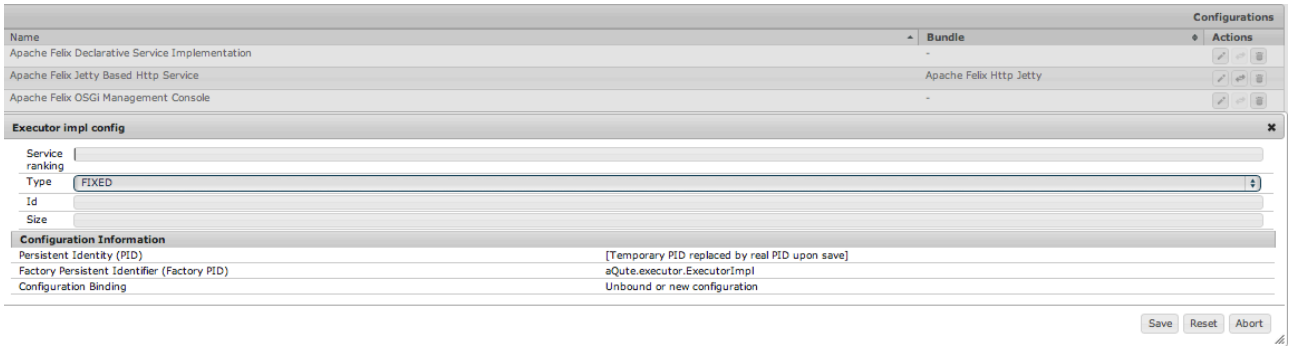
You can therefore now verify the settings through the Web Console. Go to http://localhost:8080/system/console/configMgr, you can see the following panel there (details may differ):



Clicking on the `Apache Felix Jetty Based Http Service` entry will show you a form with the details of the configuration and a means to change this. Notice that if you change this configuration manually then it will no longer be updated from the bundle. This form can sometimes be very useful to find the names of the configuration properties.

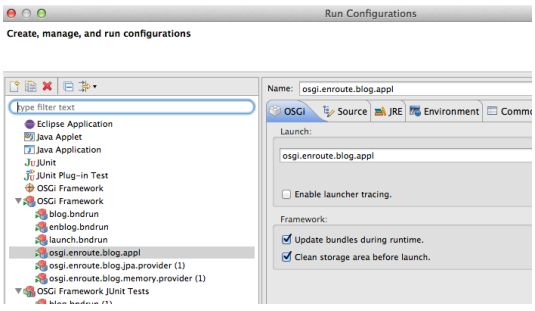


You can also create new instances for configurations that support factories. For example, it is possible to create an extra executor, just click on the + to add a new configuration or on the trash to delete one:



Tracing

During development it is quite normal that things do not work as expected. There are extensive tracing facilities build into bnd. You can activate tracing by selecting a checkbox in the `Run Configuration`. Select `Run/Debug Configurations...`, and then select the desired `OSGi Framework` configuration. On the `OSGi` tab, you find `Enable Launcher Tracing`.

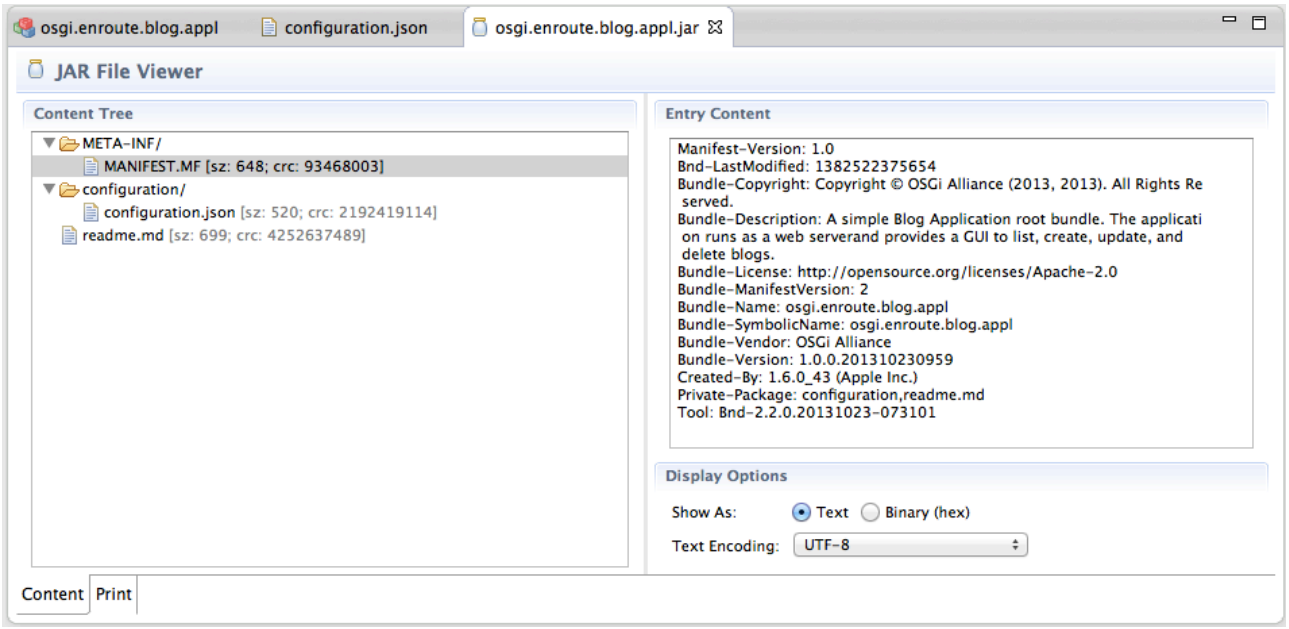


There are also a number of other options. You can also activate a number of additional traces by setting a `-runtrace` to `true` in the `bnd.bnd` file.

```
-runtrace: true
```

Content

At this moment, it is interesting to look at what is actually included in your bundle. bnd places the bundle(s) in the `generated` directory. Double click on `generated/osgi.enroute.blog.appl.jar` file. This will open the JAR viewer.



As you can see, it includes a manifest, the configuration, and the `readme.md` resource. It is interesting to look what bnd has filled in for you. Notice that you can override anything you want.

Exercise

Configure the web server to run on port 9090. You can find the configuration properties here http://felix.apache.org/documentation/subprojects/apache-felix-http-service.html.

What You Have Learned

In this lesson you've learned the following things:

- Selecting a set of bundles based on preset libraries
- Inheritance from the `cnf/build.bnd` file.
- Launching a framework
- The Gogo Shell
- The Web Console
- X-Ray
- Providing configuration information
- Tracing changes in the framework

You can verify you solution by comparing your workspace against the `01-setup` branch.

References

- bnd(tools)
  http://bndtools.org
- X-Ray
  http://softwaresimplexity.blogspot.fr/2012/05/x-rays-for-osgi.html
- Apache Felix Http & Whiteboard
  http://felix.apache.org/documentation/subprojects/apache-felix-http-service.html
- Apache Felix Web Console
  http://felix.apache.org/site/apache-felix-web-console.html
- Apache Felix Gogo Shell
  http://felix.apache.org/site/apache-felix-gogo.html
  http://felix.apache.org/site/rfc-147-overview.html
- aQute Configurer
  http://jpm4j.org/#!/p/osgi/aQute.configurer?tab=readme

3. Hello World

Goal

To create a Hello World web page.

Prerequisites

First ensure that you have checked out the setup branch in the workspace.

```
$ git checkout 01-setup
```

You should compare the the project against your own solution of the previous step. you can also continue with your previous step. If you continued from the previous phase then the framework should still be running. Verify this; if it is not running then start it. We will update the framework incrementally.

Adding a Web Page

The `aQute.webserver` bundle is an extender that provides a number of features to simplify web page applications. It most important functions is to map a directory (`static`) in a bundle to the web. All these static directories are overlaid, allowing many bundles to share the same namespace.

If a request is done, the webserver bundle will try to locate the path in one of the bundles. It will provide a number of HTTP mechanism supports like caching control, ranges, and compression. If no path is given above its servlet path then it will try to find a `/index.html` resource.

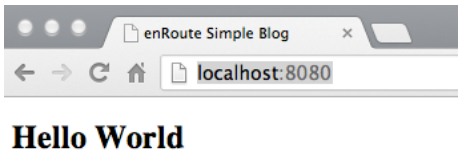
Adding a web page is therefore as simple as creating a file in the static directory. In this phase the only thing we will do is add an `static/index.html` file with the following content:

```
<!DOCTYPE html>
<html lang=en>
<head>
  <title>enRoute Simple Blog</title>
</head>
<body>
  <h1>Hello World</h1>
</body>
</html>
```

This file is a minimal but valid HTML-5 page. We must now also include the file. Since we will add more files in the following steps we add the static directory to the `-includeresource` instruction `bnd.bnd` file.

```
-includeresource: \
    {static=static}, \
    {configuration=configuration}, \
    {readme.md}
```

Again, we add the curly braces so that text files in this directory are properly pre-processed. After you saved the bnd.bnd file you can go to http://localhost:8080/ and see the following, admittedly rather boring, page:



You can make changes to the `index.html` file to test out how things work. Once you save the file, it automatically is deployed in the running framework. However, you need to refresh the browser to see these changes.

References

- HTML-5 (not this reference is to the WHATWG, not wc3!)
  http://www.whatwg.org/specs/web-apps/current-work/multipage/
- aQute Web Server
  http://jpm4j.org/#!/p/osgi/aQute.webserver?tab=readme
- bnd macros
  http://www.aqute.biz/Bnd/Macros

4. Bootstrap

Goal

To create a proper index.html page that is styled with Twitter's Bootsrap.

Prerequisites

First ensure that you have checked out the 'Hello World' branch in the workspace.

```
$ git checkout 02-hello
```

You should compare the the project against your own solution of the previous step. you can also continue with your previous step. If you continued from the previous phase then the framework should still be running. Verify this; if it is not running then start it. We will update the framework incrementally.

Background

Cascading Style Sheets (CSS) are by far the most (unnecessary) complex and mind blowing deficient web technology. In any project, the web styling can create a huge drain on productivity and, unless there are experts, results in mediocre pages. That was, until Twitter brought us Bootstrap, a CSS framework. Even though one can have serious criticism on the typography, architecture, and execution, it is in general so much better and easier to use than what technical people, like programmers, create.

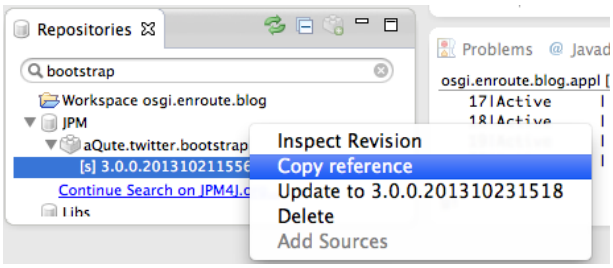
Bootstrap provides an ontology for the class attribute values. By using a fixed set of names with defined semantics and accompanied by rules to proper wrap parts in `div`'s it is possible to significantly customize web pages that look decent (though if the default Bootstrap CSS is used, it will look like thousands of other websites). Then again, Bootstrap can rathe easily be customized.

Adding Bootstrap

Bootstrap is provided as a bundle, `aQute.twitter.bootstrap`. We must add this bundle to the list of `-runbundles`. In this current workspace, we get our dependencies from JPM4j, a web site that contains all of Maven Central and more. In this workspace, there is a subset loaded, which includes, fortunately for you, this bundle with Bootstrap.

So go to the Repository view (if this is not open, you can find it in `View/Other/Repository`) and search for `bootstrap` in the search input field. This will show you the bundle in the repository.Double click the `bnd.bnd` file to select the editor and then the `Run` tab. You can drag the entry on the `Run` tab's `Run Bundles` list pane, this will then be added to the `-runbundles` instruction. However, in this case we will make use of the `Source` pane, this allows us to edit the bnd settings as text. Therefore, select the `Source` tab.

You can call up a context menu on the *revision*. The revision is the entry under the Bundle Symbolic Name with the version number. In this menu, you will find a `Copy Reference` entry. Selecting this entry creates a full reference to that revision on the clipboard.



Having this reference on the clipboard, we can now add it to the `-runbundles` instruction in the `Source` tab of the `bnd.bnd` file by pasting it. Take care to properly extend the last line with a \ and a newline. The instruction in the `bnd.bnd` file should then look like this:

```
-runbundles: \
    cnf.run.base, \
    cnf.run.web, \
    cnf.run.web.debug, \
\
    aQute.twitter.bootstrap;version='[3.0.0,4.0.0)'
```

If you save the file then the Bootstrap bundle is automatically installed in the running framework and started. You could verify this in X-Ray.

Updating the head

The first thing to address is the page encoding. In the pre-requisites of the preparation it was necessary to set the encoding of all files to UTF-8. (Eclipse has an insane default of platform dependent encoding, at least on MacOS.) The browser has no way of knowing until we tell it; we can tell it by placing a `meta` tag in the `head` tag.

```
<meta charset=utf-8 />
```

The following header provides some practical advantages, see stackoverflow. This provides a better user experience on Internet Explorer.

```
<meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1" />
```

And to support mobile browsers:

```
<meta name="viewport" content="width=device-width, initial-scale=1" />
```

The `head` tag should also contain the link to bootstrap:

```
<link href=/twitter/bootstrap/css/bootstrap.css rel=stylesheet
    media=screen>
```

The `/twitter/bootstrap/css/bootstrap.css` path is used, this is where the Bootstrap bundle has placed the files in its bundle. With the media we indicate that our page is optimized for screens. The `head` tag should now look like:

```
<head>
   <title>enRoute Simple Blog</title>
   <meta charset=utf-8 />
   <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1" />
   <meta name="viewport" content="width=device-width, initial-scale=1" />
    <link href=/twitter/bootstrap/css/bootstrap.css
          rel=stylesheet
       media=screen>
</head>
```

Layout

A layout in Bootstrap is defined of nested `div` tags marked with a number of classes to indicate their role: `container`, `row`, or cell. Cells are indicated with a given `span*` class. In this example, we create a simple container that holds a navigation bar, content, and a footer.[1] So we create an outer `div` tag to hold the container.

```
<div class="container" style="min-height: 400px">
   ...
</div>
```

We give at a minimal height to prevent the footer from sticking to the navigation bar. The remaining HTML is placed inside this `div` tag.

Navigation Bar

Bootstrap provides a navigation bar template. In this template we can brand the application (left most field in the navigation bar), add commands, and provide a search box. The following navigation bar is reversed and gives us a navigation bar with one command to create some test data.

[1] This still needs some work for bootstrap 3

```
<nav class="navbar navbar-static-top navbar-inverse navbar-default"
    role="navigation">
  <div class="navbar-header">
    <a class="navbar-brand" href="#">enRoute Blog</a>
  </div>
  <ul class="nav navbar-nav">
    <li class="active"><a href="#">Test Data</a></li>
  </ul>
  <form class="navbar-form navbar-right" role="search">
    <div class="form-group">
      <input type="text" class="form-control" placeholder="Search">
    </div>
    <button type="submit" class="btn btn-default">Submit</button>
  </form>
</nav>
```

Content

In this phase we leave the content to be 'Hello World'. For separation, we provide some extra white space.

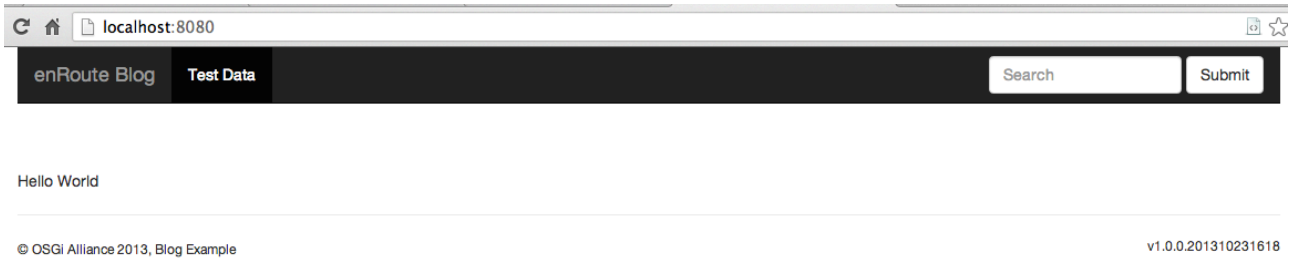
```
<div style="margin-top: 60px;"></div>
Hello World
<hr>
```

Footer

The footer is placed at the end of the page. In this case we put in a copyright and the version of our application bundle. This version is obtained using the preprocessor. Notice how we get the version on the right side with the `pull-right` class.

```
<footer>
  <p>
    <small>&copy; OSGi Alliance ${tstamp;yyyy}, Blog Example</small>
    <small class=pull-right>v${Bundle-Version}</small>
  </p>
</footer>
```

If you've saved the `bnd.bnd` file then you can go to http://localhost:8080 (do refresh) and enjoy the result:



References

- Twitter Bootsrap
  http://getbootstrap.com/
  http://getbootstrap.com/2.3.2/scaffolding.html
- aQute Twitter Bootstrap
  http://jpm4j.org/#!/p/osgi/aQute.twitter.bootstrap?tab=readme
- CSS Level 3
  http://www.w3.org/TR/css-2010/
  http://www.whatwg.org/specs/web-apps/current-work/multipage/references.html
- CSS Architecture
  http://smacss.com/book/
- Viewports
  http://www.quirksmode.org/mobile/viewports2.html
- Style sheet links
  http://www.w3schools.com/tags/att_link_media.asp

- Internet Explorer
  http://stackoverflow.com/questions/6771258/whats-the-difference-if-meta-http-equiv-x-ua-compatible-content-ie-edge-e

5. Angular

Goal

Understand the working of Angular, mainly the interaction between the HTML and the Javascript code.

Prerequisites

First ensure that you have checked out the 'Hello World' branch in the workspace.

```
$ git checkout 03-bootstrap
```

You should compare the the project against your own solution of the previous step. you can also continue with your previous step. If you continued from the previous phase then the framework should still be running. Verify this; if it is not running then start it. We will update the framework incrementally.

Background

One of the major problems in user interface applications is how to separate the different *concerns*. Smalltalk came up with the Model, View, Controller (MVC) division in the seventies of the last century; this model is still seen as the primary way to architect user interface applications. Unfortunately, most MVC implementations fall far short since they almost always require some code to update the view when the model changes. In a pure MVC model, no code would be necessary, just setting a variable in the model should suffice. Angular gets awfully close to this idea.

Angular detects changes in the model's values and automatically updates views that reference that value; automagically. Additionally, Angular is highly modular with its service model and the possibility to use HTML fragments, filters, and directives.

In this phase we will create a small (trivial) Blog Application in Javascript. It shows a form with an editor for a blog post. It also shows a list of blog posts. Items in the list can be deleted.

Adding Angular

We first have to add Angular. Just like Bootstrap, Angular is provided as a bundle: `aQute.angular.stable`. You can find this bundle in the JPM repository. You should also add this bundle to the `-runbundles` in the `bnd.bnd` file, either as text, via drag and drop, or the + on the `Run` tab's `Run Bundles` list pane.

The **`-runbundles`** instruction should look like:

```
-runbundles: \
    cnf.run.base, \
    cnf.run.web, \
    cnf.run.web.debug, \
\
    aQute.twitter.bootstrap;version=3.0.0, \
    aQute.google.angular.stable;version=1.0.8
```

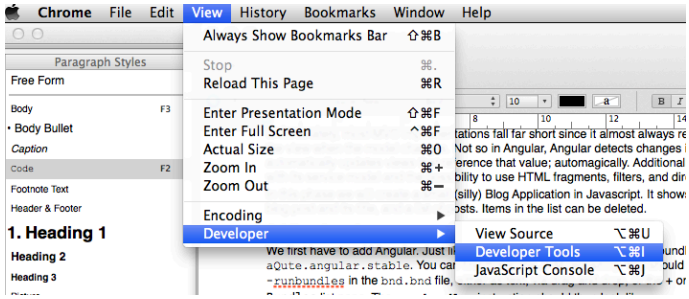
Adding this bundle makes Angular available in an HTML page under the `/google/angular/angular.js` path. We add the following line to the `index.html` file at the end (just before the `body` tag is closed).

```
    <script type="text/javascript" src="/google/angular/angular.js"></script>
  </body>
</html>
```

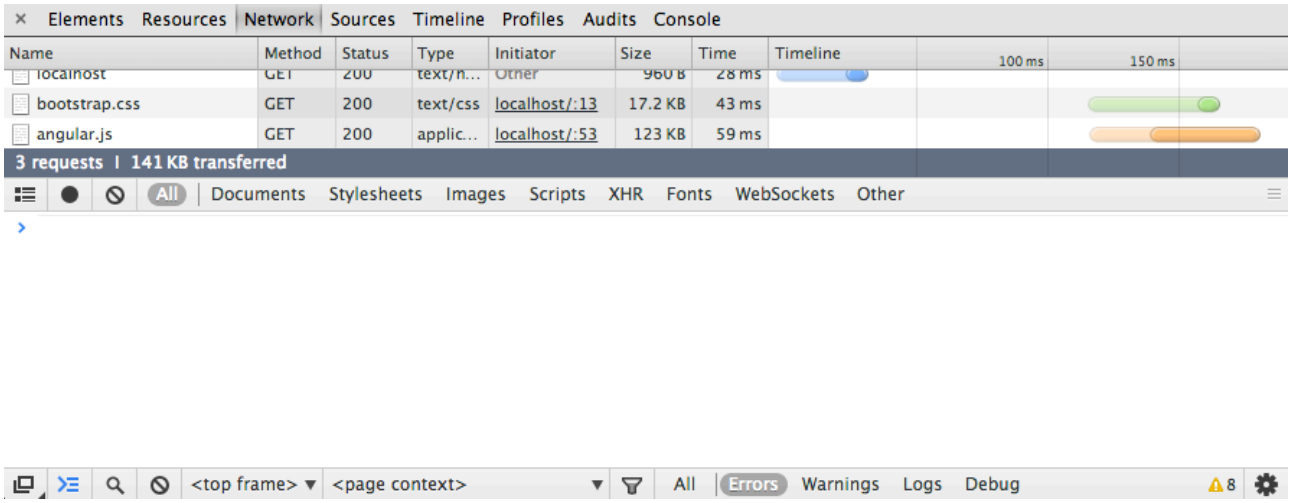
Angular is loaded at the end so that the HTML is already visible while it is loaded. Otherwise, a slow connection could make the page feel very unresponsive.

Debugging Javascript

Since we are now starting to work with Javascript and CSS make sure you understand the support your browser gives in debugging Javascript and CSS. All browser have this support. In Google Chrome you find a `View/Developer/Developer Tools` menu entry.



This opens a window at the bottom of the page. Make sure you have the Javascript console open. Angular has extensive error checking. It is a utter waste of time wondering why your application does not work while hidden in the Javascript console there is an exception displayed that perfectly tells you why. In Chrome you can open the Javascript console by clicking on the second icon left under.



Whatever browser you are using, make sure you understand its debugging support. You can safe untold hours by watching the log, checking the network transfers, and inspecting DOM elements.

Minimal

To get started, we need to tell Angular that it can touch the DOM of the page, this requires an `ng-app` attribute at a tag, any content inside that tag is then free game for Angular. In our case, we add the `ng-app` attribute at the `html` tag since the whole page is an Angular application.

```
<html lang="en" ng-app>
```

In the previous step we added a Test Data button in the navigation bar. Change this so that it sets the `hello` variable in the *model* by adding an `ng-click` attribute. An `ng-click` attribute specifies an instruction to execute when the tag is pressed. When you press the `Test Data` button, we set the hello variable to `'OSGi'`.

```
<li class="active"><a ng-click="hello='OSGi'">Test Data</a></li>
```

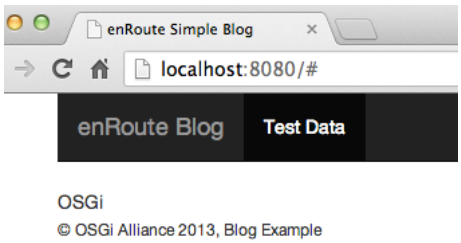
Though the syntax looks Javascript code it is actually not Javascript, it is more restricted but also provides many extra features; Angular actually has its own compiler.

Any variables referred in this context are stored in the current *scope*. A scope is a Javascript variable that can be linked to a parent scope and contains the variables of the applications model as properties. In this case we only have a *root scope*, this one is set by the `ng-app` attribute.

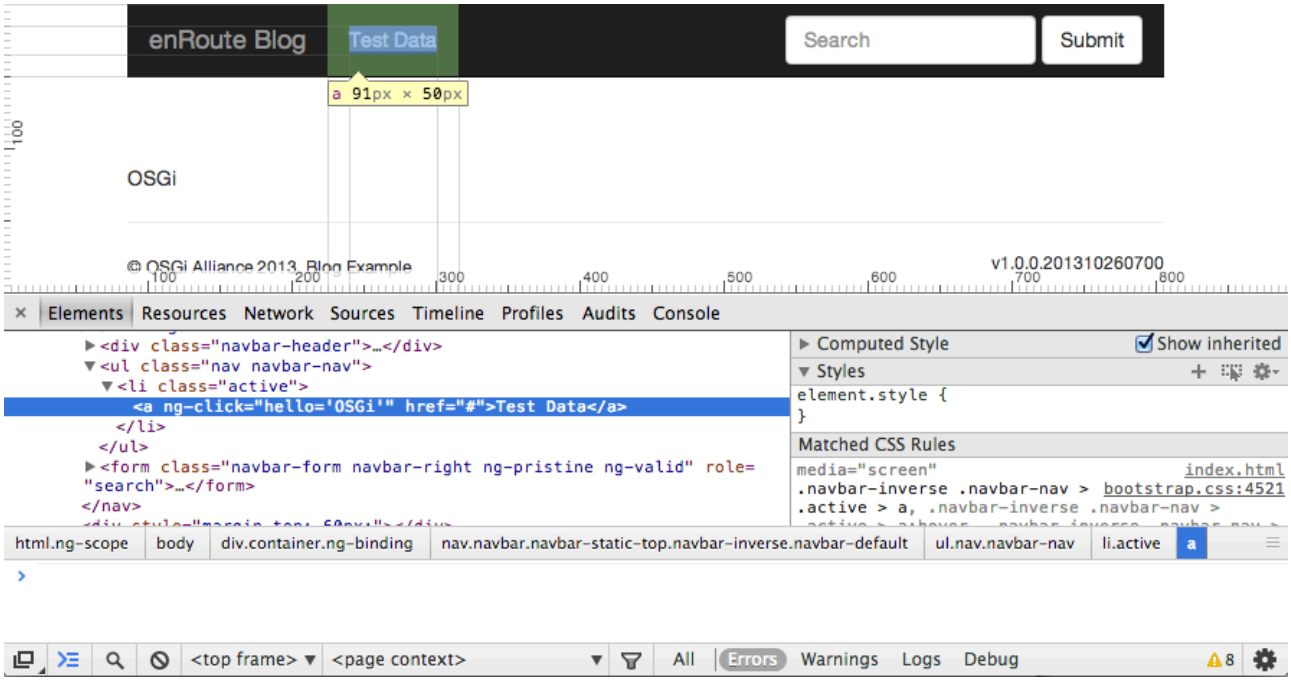
We can refer to the `hello` variable anywhere on the HTML page. The simplest way to see the value of this variable is to use the double curly braces syntax , like`{{hello}}`. You can insert this in place of the 'Hello World' text.

```
</div>
{{hello}}
<hr>
```

If you refresh the page now and click on the Test Data button in the navigation bar you will see OSGi where it used to say 'Hello World'.



If you need to debug, use the `Inspect Element` of your browser to see what is set in the tags on the page. You can do this by selecting for example the Test Data button with your mouse and calling up the context menu. Most browsers today allow you to inspect the current element. Angular provides extensive information in the DOM. For example in Chrome:



Module

So far we have used the *demo* mode, useful to show some principles but useless for real applications. Real applications tend to become quite complex and therefore require a way to break into different, separate parts. A.k.a. modularity.

To create our own module we first need to create a Javascript file and load it. Add a `static/enroute/blog/js/blog.js` file to the project and load this from the `index.html` file. Begin with a `blog.js` file that contains:

```
(function(angular) {
    var MODULE = angular.module("Blog", []);
})(angular)
```

The change to the index.html file looks as follows:

```
    <script type="text/javascript" src="/google/angular/angular.js"></script>
    <script type="text/javascript" src="/enroute/blog/js/blog.js"></script>
  </body>
```

This is the skeleton of a Javascript module, that registers a module `Blog` with Angular. A Javascript module is a function, providing private variables inside the function. This function is anonymous, it is created and directly called once, with `angular` as parameter.

This parameter is then used to create an Angular module, stored in the private MODULE variable so it is available to anyone in the module. The list given as second parameter is required, it specifies the names of any modules we depend on.

The next step is to associate the index.html page with this module. This is done by setting the `ng-app` attribute to the name of the module.

```
<!DOCTYPE html>
<html lang="en" ng-app=Blog>
<head>
```

Try this out, the behavior (showing 'OSGi' when you click `Test Data`) should still work.

Controller

The next part consists of adding a *top level controller*. A controller provides the business logic of the application. A controller is associated with a specific DOM element and provides the behavior for anything that happens inside that DOM element; controllers can, and will, be nested.

A controller is a Javascript function that receives the model, which is called the `$scope`. This name must be used exactly as here since Angular injects these variables based on how they are called in the function invocation! To see how this works, we create a top level controller, associate it with the `body` tag, and assign an initial value to the `hello` variable we used in the `Test Data` button.

```
(function(angular) {
    var MODULE = angular.module("Blog", []);
    window.enBlog = function($scope) {
           $scope.hello = 'Java';
    }
})(angular)
```

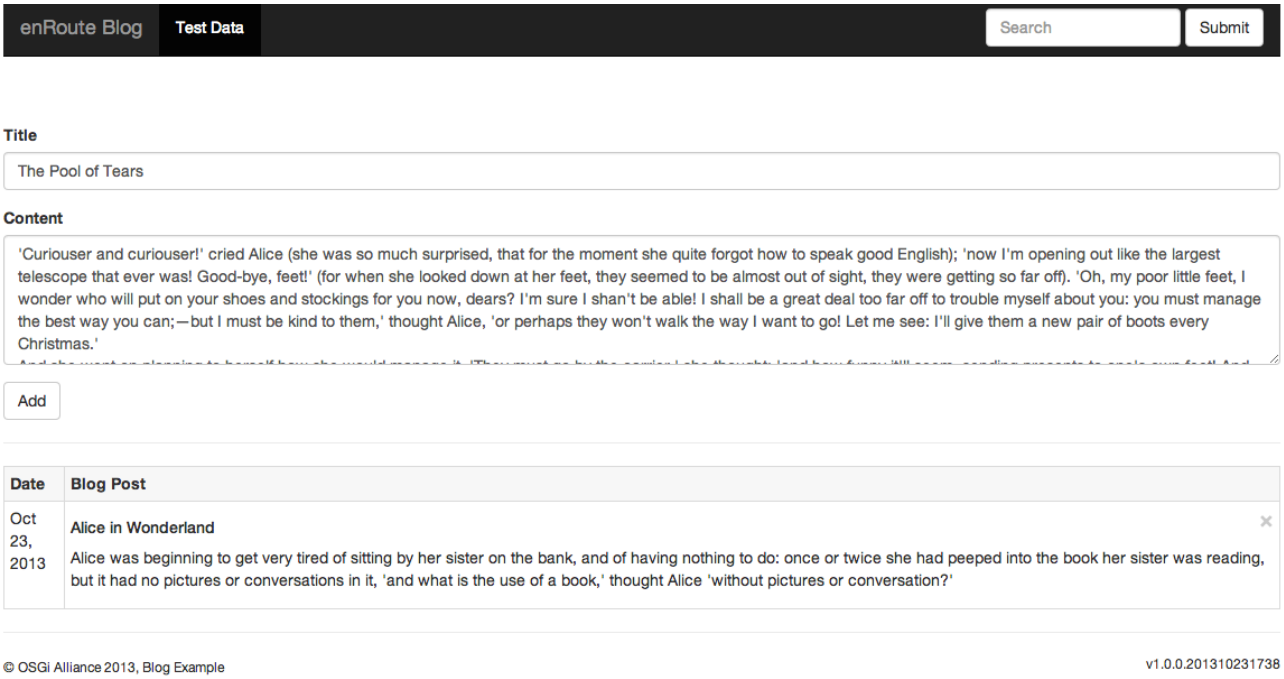
We can associate the top level controller at the `body` tag of the `index.html` file. We can use the Javascript name, `enBlog` since we made it a global variable. (In Javascript all `window`'s properties are global variables.)

```
</head>
<body ng-controller=enBlog>
  <div class="container" ...
```

If we run the application now, the value 'Java' will initially be associated with the `hello` variable. If we press the Test Data button, it will be set to 'OSGi'. We have now shown that we can communicate between the HTML and the Javascript model.

Blog Application

We now create a very simple application in Javascript. It consist of an editor of a blog post (title and content), and a save button that pushes it on a list of posts. In the lists, the posts can be removed by clicking a delete button. Something like this.



So let's design this display. The following html, which should replace the `{{hello}}` reference, shows the form with the title and content edit fields:

```
<form>
  <div class="form-group">
    <label>Title</label>
    <input type="text" class="form-control"
       placeholder="Enter title">
  </div>
  <div class="form-group">
    <label>Content</label>
    <textarea class="form-control"
       placeholder="Content"></textarea>
  </div>
  <button type="submit" class="btn btn-default">Add</button>
</form>
```

The next part is to associate this form with the model. In Angular, in general the model is specified with the `ng-model` attribute, this is a bi-directional link. We have two fields here, the title and the content. Lets associate them with the `title` and `content` variables.

```
<input ng-model=title type="text" class="form-control"
       placeholder="Enter title">

<textarea ng-model=content class="form-control"
       placeholder="Content"></textarea>
```

The last part we need is an event to save the edited text. This action needs to be associated with the button, as we've seen before, the `ng-click` attribute is associated with actions.

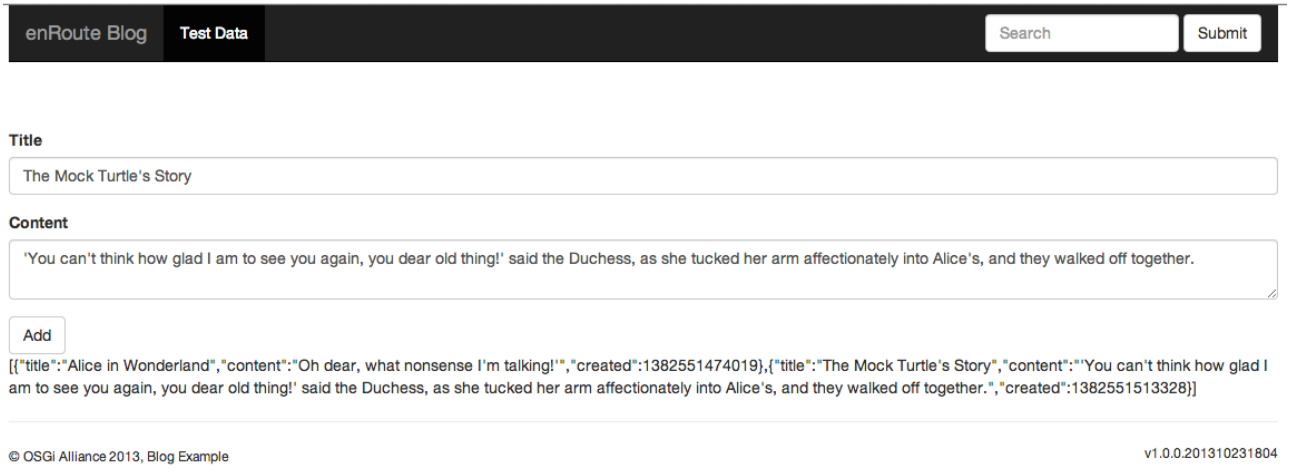
```
<button ng-click=save() type="submit" class="btn btn-default">Add</button>
```

This last change requires a function `save()` in the current `$scope`. We can add this in our enBlog controller since it manages the scope for the whole HTML page. So we can add a `save` function that pushes an object with a `title` and a `content` property on a `posts` list.

```
(function(angular) {
  var MODULE = angular.module("Blog", []);
  window.enBlog = function($scope) {
    $scope.posts = [];
    $scope.save = function() {
      $scope.posts.push( { title: $scope.title, content: $scope.content,
        created: new Date().getTime() } );
    };
  }
})(angular)
```

This would be a good moment to see where we are, even though we have not implemented the table with saved posts yet. There is an easy way to see the posts (without any formatting) by just placing a `{{posts}}` somewhere in the page. This shows the list in JSON format. So you can add this at the end of the form.

Save the pages, go to http://localhost:8080/# and refresh the browser! Type some text and press the add button.



Viewing the Blog Posts

The next stage is to view the lists of blog posts. Tables are still eminently suitable for that. You can add the following fragment after the form. This is a simple table. showing the three fields of the blog post.

```
<div>
  <hr>
  <table class="table table-bordered table-striped table-condensed">
    <tr>
      <th>Date</th>
      <th>Blog Post</th>
    </tr>
    <tr>
      <td>post.created</td>
      <td>
        <button class='close pull-right'>&times;</button>
        <h5>post.title</h5>
        <p>post.content
      </td>
    </tr>
  </table>
</div>
```

This displays:

If you try out this change you will see that there is only one row and that the text shows the names of the variables: `post.created`, `post.title`, and `post.content`. In this example, we have not yet shown where the `post` variable comes from.

Angular can repeat a tag multiple times with the `ng-repeat` attribute. This attribute takes a *comprehension* expression. There are quite a few variations, but in its simplest form it looks like: `variable in expression`. Where `variable` is the user defined loop variable and expression is an angular expression giving the collection to enumerate over. In our case: `post in posts`.

The `tr` tag needs to be repeated since it holds the row with the blog post information. We also need to change the variable names to show their value, i.e. add the curly braces.

```
<div>
  <hr>
  <table class="table table-bordered table-striped table-condensed">
    <tr>
      <th>Date</th>
      <th>Blog Post</th>
    </tr>
    <tr ng-repeat="post in posts">
      <td>{{post.created}}</td>
      <td>
        <button class='close pull-right'>&times;</button>
        <h5>{{post.title}}</h5>
        <p>{{post.content}}
      </td>
    </tr>
  </table>
</div>
```

Filters

If you looked careful you will have seen that the date in the list is a number, it is the number of milliseconds since Jan 1 1970. Obviously this is not very useful for mere mortals. We could create a function in the controller that converted it. However, Angular has special support for these cases, they are called *filters*. Angular expressions can be piped, just like in unix, through filters. Out of the box, Angular supports a number of filters to sort lists, filter lists, handle currency formatting, limiting, number formatting, casing, and of course date handling. Filters can have parameters but in this case the default date format works well:

```
<tr ng-repeat="post in posts">
  <td>{{post.created | date }}</td>
```

Modules can actually also add new custom filters.

Test Data

It is a bit tedious to add text in the title and content fields all the time. So lets add a function that adds some test data. Let us first call this function from the `Test Data` button in the navigation bar.

```
<ul class="nav navbar-nav">
  <li class=active>
    <a ng-click=testdata()>Test Data</a>
  </li>
</ul>
```

We add this function in the `enBlog` controller since we need to access the `title` and `content` variables, we will just set them to an example value. So add the following function to the enBlog controller body (after the `save` function):

```
$scope.testdata = function() {
  $scope.title = EXAMPLE.title;
  $scope.content = EXAMPLE.content;
}
```

We are referring to a constant, the `EXAMPLE` variable. Make sure you use a `var` keyword, to keep the variable local to this module, we call these *module constants*. Let's add this to the top.

```
var MODULE = angular.module("Blog", []);
var EXAMPLE = {
  title : "Alice in Wonderland",
  content : "There was a table ..."
};

window.enBlog = function ...
```

Of course you can write your own text. It is actually advised to make the content much, much longer.

Now clicking on the Test Data button will fill the `input` field and the `textarea` tags with content. Making it easier to add a number of entries.

You might have noticed that the cursor did not indicate you could press the `Test Data` button in the navigation bar. In general, any tag in Angular that has an `ng-click` attribute should have a cursor indicating that it is clickable. This can be achieved with the following CSS in the `head` tag.

```
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <style>[ng-click] { cursor: pointer; }</style>
  </head>
```

Deleting an Entry

In the table we are showing a × (`&times; \u2A09`) symbol on the right top to delete an entry. However, for now this entry is not deleted. So lets add this behavior. First add the function call to the button. Call the function `remove`, and pass it the `post` as argument.

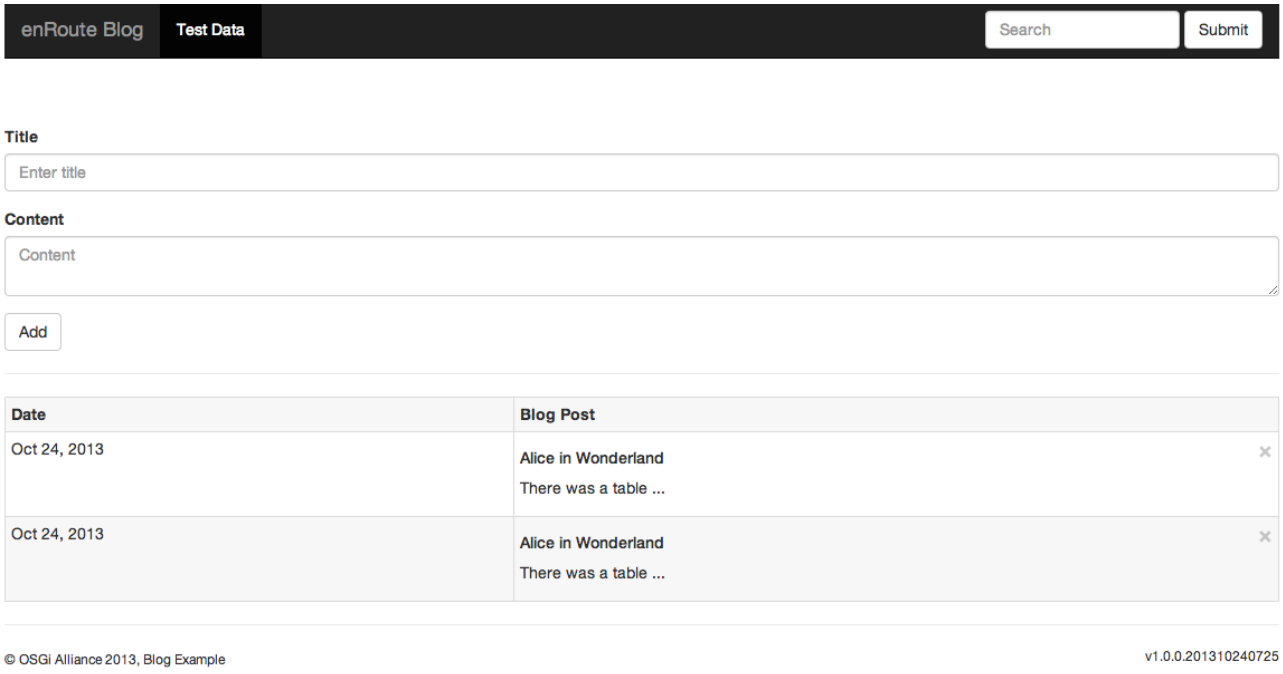
```
        <td>
          <button ng-click="remove(post)"
               class='close pull-right'>&times;</button>
          <h5>post.title</h5>
```

We now need to add the `remove` function to the controller after the `save` function. Weirdly, removing an element of a list is a bit convoluted in Javascript. You first get the index of the element and then splice the array on that position. This splice function is very powerful, we just use it to delete 1 element after the found index.

Since we've changed the controller quite a bit, we repeat the whole controller here.

```
  window.enBlog = function($scope, $location, $route) {
    $scope.posts = []

    $scope.save = function() {
      $scope.posts.push({
        title : $scope.title,
        content : $scope.content,
        created : new Date().getTime()
      });
      $scope.title = $scope.content = "";
    }

    $scope.remove = function(post) {
      $scope.posts.splice($scope.posts.indexOf(post), 1);
    }

    $scope.testdata = function() {
      $scope.title = EXAMPLE.title;
      $scope.content = EXAMPLE.content;
    }
  }
```

Time to try it out http://localhost:8080/ ... Don't forget to refresh the browser.



References

- Angular
  http://docs.angularjs.org
  http://docs.angularjs.org/api/ng.directive:ngRepeat
  http://docs.angularjs.org/guide/dev_guide.templates.filters.using_filters
- Javascript
  http://www.w3schools.com/jsref/jsref_splice.asp
- Chrome Debugging JS
  https://developers.google.com/chrome-developer-tools/docs/javascript-debugging

6. REST Calls

Goal

Communicate to the back-end server through REST calls.

Prerequisites

First ensure that you have checked out the following branch in the workspace.

```
$ git checkout 04-angular
```

You should compare the the project against your own solution of the previous step. you can also continue with your previous step. If you continued from the previous phase then the framework should still be running. Verify this; if it is not running then start it. We will update the framework incrementally.

Background

So far, most of the effort has been in the `index.html` page and the `blog.js` Javascript, both executed in the browser. Though the delivery of Bootstrap and Angular through OSGi bundles is convenient, it is not exactly the killer application for OSGi. To build a killer application need to communicate between the Javascript program running in the browser and a service in the OSGi framework to get there.

There are a number of solutions to communicate. In this tutorial we use REST, mostly because it is very well supported in Angular with the `$resource` service.

The `$resource` service provides an object oriented framework to perform Create, Read, Update, and Delete operations (CRUD). The `$resource` service is a function that can create a *factory* for REST objects. *Resource* objects are then created with that factory with `new`. Methods on the resource objects then provide

the CRUD operations. There are also a number of factory methods on the resource factory to query the server, get instances, save an instance, and remove an instance. This is all a bit weird for a Java programmer but it does work.

Another difficult aspect (for Java developers) of this REST framework is that REST calls return objects that are empty and that are only later, asynchronously, filled with their properties. That is, returned values must be an object, primitive values like numbers and strings do not work well since these do not have properties that can be provided later.

On the back-end side, in OSGi, we need to provide a *REST endpoint*. This endpoint receives the URL with its parameters and an optional body and must execute the desired function. In this exercise we use the aQute.rest.srv bundle. This bundle tracks `ResourceManager` services. A Resource Manager is a service that offers a number of public methods that map to REST endpoints. A method is an endpoint when:

- The name starts with one of the HTTP verbs: `get`, `post`, `put`, `delete`, `option`, `head`, ...
- The part of the name after the verb starts with an upper case character is the REST URI path (the REST server provides an additional prefix).
- The first argument extends the `Options` interface. The `Options` interface can provide access to the body, the parameters, the servlet request, and the servlet response. The body of the request can be accessed in a type-safe way by extending the Options interface and implementing the `_()` method, the return type will be used to coerce the body in from a JSON package.
- Additional arguments are the parameters in the request URI; they can be of any type and are converted from the URI. To access the parameters in a type safe way, implement methods with the name of the parameter on an interface that extends the `Options` interface, the return type of this method is used to coerce the parameter.

For example, the method `BlogPost getBlogpost(Options options, long id)` maps to:

```
GET /rest/blogpost/123 HTTP/1.1
```

You can create the following interface:

```
interface SaveOptions extends Options {
  BlogPost _();
  int limit();
  List<Person> authors();
}
```

Then the method `BlogPost postBlogpost(SaveOptions options,long id)` maps to:

```
POST /rest/blogpost/123 HTTP/1.1
```

All transfers are done in JSON. The public fields in the Java types are used to drive the encoding and decoding. So the Java code can stay type safe and does not have to convert from the limited set of JSON data types.

Add Dependency to the $resource Module

The `$resource` service comes from the `ngResource` module, it is not built into Angular. We need to make some changes to make this new module available to our own `Blog` module. The first change is in the `index.html` file, we need to load the module. Since it is included in the the Angular bundle, we can easily add it:

```
    <script type="text/javascript" src="/google/angular/angular.js"></script>
    <script type="text/javascript" src="/google/angular/angular-resource.js">
        </script>
    <script type="text/javascript" src="/enroute/blog/js/blog.js"></script>
</body>
```

The order after the Angular include is not relevant. Angular will just record the modules during load time, only after everything is loaded will it find out dependencies and initialize the modules. It is therefore necessary to explicitly declare module dependencies. The previously mysterious list in `blog.js`'s module declaration must now be filled in.

```
(function(angular) {
    var MODULE = angular.module("Blog", ['ngResource']);
```

And last, we need to add the `$resource` service in the enBlog controller's prototype:

```
window.enBlog = function($scope, $resource) {
```

Angular injects these services by the name they have in the prototype of the function (it uses a function's `toString` method to find out these names).

So far we have only added the dependencies, this has not changed any behavior yet.

Test Data Front End

In the previous step of this tutorial we created the test data in the Angular controller, using a constant in the module. Let's get this test data from the server instead.

We call this resource type `Command`, since it is likely that we will have more such commands that do not map very well to the REST URI, we can use this factory then as a catch-all. This factory must be available everywhere in our module, we therefore need to declare it ahead of time. In Angular you cannot just initialize when you create the module (this happens at load time), you must wait until Angular has everything setup and calls you back. We will therefore create the `Command` resource factory in the `enBlog` controller. However, first add the `Command` variable so it will not become a global variable, we need it to be a module variable:

```
var MODULE = angular.module("Blog", ['ngResource']);
var Command;
```

The URI we associate with this resource type is:

```
/rest/command/<command>
```

The `<command>` part of the URI will reflect an enum since it is likely we will have multiple commands. The following is the code you can add in the `enBlog` controller's body, it will create the `Command` factory.

```
window.enBlog = function($scope, $resource) {
  Command = $resource("/rest/command/:command");
  $scope.posts = []
```

The `:command` part of the given URI matches the enum we discussed earlier, it must be provided as a property of an object given as parameter. So let's call this `TESTDATA`. We can now replace the `testdata` function in the `enBlog` controller with the following:

```
$scope.testdata = function() {
  $scope.post = Command.get({command : 'TESTDATA'});
}
```

The `Test Data` button will create a `GET /rest/command/TESTDATA` HTTP/1.1 request. To see the result we could display the `post` variable in the `index.html` file. Note that Angular will return a place holder in the `post` variable and update this place holder with the actual properties once the object arrives. Until that moment, it will just not be displayed. However, this takes a fraction of a second.

```
</nav>
{{post}}
<form role="form">
```

Obviously, this will not work yet since the back end is not written yet, nor have we activated the REST server.

Starting the REST server

The aQute.rest.srv bundle requires a configuration record to start its service. We therefore need to add it to the `configuration/configuration.json` file. Make sure the JSON format of the overall file is correct. The parser is very sensitive to quotes and commas, errors in parsing are reported in the log. You can always verify your JSON online.

```
        "size" : 16
    }, {
        "service.factoryPid" : "aQute.rest.impl.servlet.RestServlet",
        "service.pid" : "RestEndpoint",
        "angular" : true,
        "alias" : "/rest"
    }
]
```

This creates a REST server on /rest with Angular support. The angular flag indicates support for preventing Cross Site Scripting attacks.

Once you save the configuration you can check http://localhost:8080/system/console/configMgr. The `Rest servlet config` should have a factory configuration. You can also check X-Ray. If you hover over the `aQute.rest.srv` bundle you see that the LED of the component is now green (it was red before you saved the configuration. The component now registers a `Servlet` service (which is picked up by the Apache Felix Whiteboard support) and it listens for `ResourceManager` services (abbreviated to `RsrcMngr`), note the dotted outline of the service.

Back End

Finally we're reaching Java code! We have to create an OSGi service that has the method `Object getCommand(Options options, Command command)`. So create a new class in the `osgi.enroute.blog.appl` package called `BlogApp`. This class should look like:

```
@Component
public class BlogApp implements ResourceManager {
  enum Command { TESTDATA }

  public static class BlogPost {
    public Stringcontent;
    public Stringtitle;

    BlogPost(String title, String content) {
      this.title = title;
      this.content = content;
    }
  }

  public Object getCommand(Options opts, Command command) throws Exception {
    switch (command) {
      case TESTDATA :
        BlogPost bp = new BlogPost("A Mad Tea Party", "'You can't ...");
        return bp;
    }
    return null;
  }
}
```

This class imports the `ResourceManager` class and the `Options` interface, however, they are not on the *build path*. This is the reason why you get the errors.

The build path is maintained in the `bnd.bnd` file, in the `-buildpath` instruction. The missing classes are in the `aQute.rest.srv` bundle, you can find this bundle in the JPM repository, just search for it in the repository view. The `@Component` annotation comes from bnd and can be found in the `biz.aQute.bnd.annotation`[2] bundle, also available in the JPM repository.

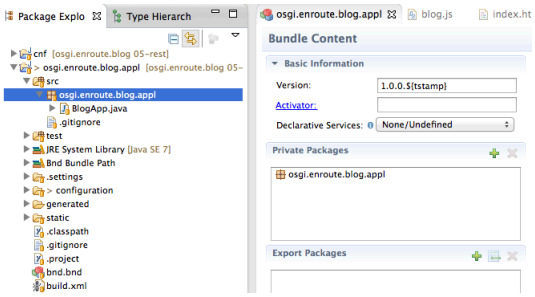
Double click on the `bnd.bnd` file and select the `Build` tab. You can drag the revision to the `Build Path` list panel. Alternatively, you can also add it manually (maybe using the `Copy Reference` context menu entry). Select the `Source` tab, and add the following:

```
-buildpath: \
    aQute.rest.srv;version=2.0, \
    biz.aQute.bnd.annotation;version=2.2
```

In the `BlogApp` class we create an enum to hold our different commands, so far only `TESTDATA`. Then we create a static class to hold our Blog Post. Its public fields map 1:1 to the Javascript object in the browser. In the `getCommand` method we `switch` on the given command. The REST server automatically converts the last segment of the URI (the string 'TESTDATA') to the enum `TESTDATA` as an argument. The method then switches on the `enum`'s value and returns a `BlogPost` object.

However, before this works, we need to add the package to the bundle. This can be done by selecting the `bnd.bnd` file's `Contents` tab, and dragging the package into the `Private Packages` list pane. Don't forget to save.

[2] These annotations will be replaced with the corresponding OSGi annotations. However, so far the bnd annotations provide some extra features.



Alternatively, you can check the `Source` tab and add it manually:

```
Bundle-Description:     \
 A simple Blog Application root bundle. The application runs as a web server\
 and provides a GUI to list, create, update, and delete blogs.

Private-Package:        \
    osgi.enroute.blog.appl
```

After saving your bundle's component you should be attached by the aQute.rest.src bundle, you can check X-Ray or refresh your browser and click `Test Data` in the navigation bar. This should show you the test output.

{"content":"'You can't ...","title":"A Mad Tea Party"}

Setting the Variables

In reality, we do not need a `post` variable, we need to set the `title` and `content` variables. However, when we call the `get` method on the `Command` resource factory we get a place holder back, the actual properties are not set until much later, when the server has returned its response. How do we get an event when this finally happens?

The prototype of the `get` method actually allows a callback to be specified as the second argument. We can therefore modify the `testdata` function as follows:

```
$scope.testdata = function() {
  $scope.post = Command.get({command : 'TESTDATA'},
    function(post) {
      $scope.title = post.title;
      $scope.content = post.content;
    }
  );
}
```

You can remove the `{{post}}` from `index.html` that was added to show the output since we now update the `input` field and the `textarea` for the `title` and `content` variables. Time to test!

Error Handling

We totally ignored errors so far. Obviously errors are important to handle, the internet can be reliable unreliable. It turns out that the `get` method on the `Command` resource factory can take a third argument: an error function. We could handle the errors inline all the time but we likely have to make quite a few rest calls in the remainder of this tutorial. Handling errors in all these cases is cumbersome.

One solution is to create two module global functions: `ok` and `error`. These functions could maintain a message at the top of the screen. They can then be passed to the REST calls as parameters. In preparation, we make these functions available to the whole module:

```
var MODULE = angular.module("Blog", ['ngResource']);
var Command, error, ok;
```

In the `testdata` function we now create the `error` and `ok` function that set and clear a `message` variable.

```
error      = function(result) { $scope.message = "[" + result.status + "]"; }
ok         = function(result) { $scope.message = ""; }
```

It would be nice to see this `message` variable so we need to change the `index.html` file.

```
  </form>
</nav>
<div>{{message}}</div>
```

We can use Bootstrap to turn this into an *alert*. This requires the class `alert` in the class attribute as well as a class specifying level, either: `alert-warning` in our case.

Angular has an `ng-class` attribute that we can pass a special object. Its property *names* are CSS class names and its value is a `boolean` Angular expression (well for Javascript everything is a `boolean`) that indicates if the given property/CSS-class name should be present or not in the tag's `class` attribute. With this information we can make the message more clear when there is an actual error:

```
<div class=alert ng-class="{'alert-warning':message}">{{message}}</div>
```

And adapt the `testdata` function to use our `ok` and `error` function.

```
$scope.testdata = function() {
  $scope.post = Command.get({command : 'TESTDATA'},
    function(post) {
      $scope.title = post.title;
      $scope.content = post.content;
      ok();
    }, error);
}
```

Notice that we call the module global `ok()` function in our own function, while the `error` function is passed as an object.

How do we test this? Why not add a method that throws an Exception on the server, this will allow us to test the whole chain. Just add the following function after the `testdata` function:

```
$scope.exception = function() {
  Command.get({command : 'EXCEPTION'}, ok, error);
}
```

And change the `index.html` file:

```
<ul class="nav navbar-nav">
  <li class="active"><a ng-click=testdata()>Test Data</a></li>
  <li class="active"><a ng-click=exception()>Exception</a></li>
</ul>
```

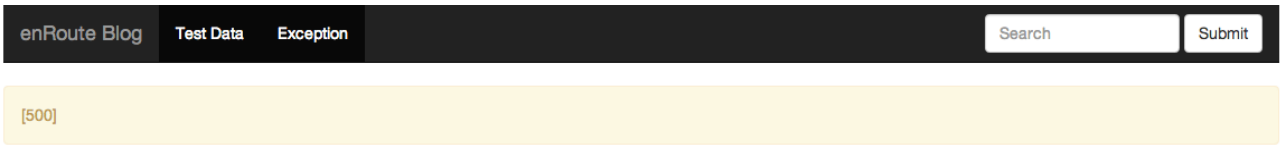
Last, but not least we need to change to the `BlogApp` class. First we add a new `EXCEPTION` field in the `Command` enum.

```
enum Command { TESTDATA, EXCEPTION }
```

Then we have to add a new case for this `EXCEPTION` enum in the `getCommand` method.

```
                case EXCEPTION :
                      throw new Exception("Yuck!");
```

This will result in a server error when you press the exception button.



References

- JSON Online checker
  http://jsonlint.com/
- Angular $resource service/module
  http://docs.angularjs.org/api/ngResource.$resource
- Angular Security Considerations
  http://docs.angularjs.org/api/ng.$http#description_security-considerations

- aQute.rest.srv
  http://jpm4j.org/#!/p/osgi/aQute.rest.srv?tab=readme
- Bootstrap Alerts
  http://getbootstrap.com/components/#alerts

7. Blog User Interface

Goal

Design a web-based user interface that can grow into a complex application.

Prerequisites

First ensure that you have checked out the following branch in the workspace.

```
$ git checkout 05-rest
```

You should compare the the project against your own solution of the previous step. you can also continue with your previous step. If you continued from the previous phase then the framework should still be running. Verify this; if it is not running then start it. We will update the framework incrementally.

Background

The last time we worked with Angular we developed a *toy* application. The primary purpose was to provide insight how the `$scope` variables interacted with the HTML. However, the approach used will soon fall apart if you try to build a larger application.

The `index.html` file would soon become unwieldy if we tried to cram all HTML in there. Angular therefore provides support for *templates*. Templates are HTML fragments that can be included anywhere in an HTML page, including in other templates. The actual template can be chosen by a variable. Templates make it possible to build the application out of small building blocks.

There is also another aspect. All state of the application was maintained in Javascript. Many AJAX applications are build like that unfortunately. If you maintain all state in Javascript then you will have a hard time supporting forward and backward browser functions as well as bookmarking. Javascript applications should use the URI of the page to make the application navigable. Support for this model is excellent in Angular, although slightly hidden in the documentation. In Angular, it is possible to setup a routing table that maps a URI to an HTML fragment and a controller for that fragment using a pattern match.

Routing

So lets design the URI space. The URI space for a blog application could look like:

URI	Controller	Template	Description
/	Home	home.htm	home, list all blogs
/post/:id	PostView	post/view.htm	view the blog post with :id
/edit	PostNew	post/edit.htm	edit a new blog post
/edit/:id	PostEdit	post/edit.htm	edit an existing blog post
/search?query=	PostSearch	post/search.htm	list all blogs matching the query

It is time to design the data model since the fragments will refer to this, using a consistent set of names saves much misery later.

Variable	Type	Description
posts	[] list	A list of blog posts
post	{} object	The current blog post
edit	{} object	A blog post being edited
search	() function	Will be discussed later

Home Page

Angular has excellent URI routing support with the builtin `$route` service. This service can be configured during a module's configuration (you configure the `$routeProvider` service then, the `Provider` suffix is special), it will then watch the URLs that are visited through the `$location` service and install an HTML template based on the configured routing table (different modules can add more entries).

Let's install a route for the home table and add its controller `Home`:

```
var Command, error, ok;
MODULE.config(
  function($routeProvider) {
    $routeProvider.when('/', {
      templateUrl : '/enroute/blog/htm/home.htm',
      controller  : Home
    });
    $routeProvider.otherwise({redirectTo : '/'});
  }
);
function Home($scope) {}
```

We must now add an HTML fragment in `static/enroute/blog/htm/home.htm` file. Just to see how this works, create this file and put "Hello World" in it.

The HTML templates from the routing table are inserted at a tag that has the `ng-view` attribute, general a `div` tag. We therefore have to modify the `index.html` file. We must remove all the form and the list view tags we did before and insert the `ng-view` attribute between the alert message and the footer.

```
<div class=alert ng-class="{'alert-warning': message}">{{message}}</div>
<div ng-view></div>
```

It should now work again ... so test http://localhost:8080/#/ and enjoy your welcome.

Blog Post View

To view a Blog Post we need to show the content, the title, and provide some buttons to let it be edited and deleted. Adding this functionality will require the following actions:

- Add a route from `/post/:id`
- Add a PostView controller
- Add a template file in `static/enroute/blog/htm/post/view.htm`.

You will see this pattern frequently in designing a web application. Actually, you normally also have to add support for Javascript testing, the back-end code, and the back-end test. In Ruby, they have Cucumber files that make it easy to keep all these details spread out over many files together.

So step by step, first the route in `blog.js`.

```
$routeProvider.when('/', {
  templateUrl : '/enroute/blog/htm/home.htm',
  controller  : Home
});
$routeProvider.when('/post/:id', {
  templateUrl : '/enroute/blog/htm/post/view.htm',
  controller  : PostView
});
$routeProvider.otherwise({redirectTo : '/'});
```

The controller should support the following functions.

- Fake a Blog Post object.
- An `edit` function, to be associated with the `Edit` button for the selected post.
- A `delete` function, to be associated with the `Delete` button for the selected post.

In this controller we need access to the `id` of the post, which is part of the URI. You can get this as a property on the `$routeParams` service, so this service needs to be injected in the prototype of the controller's function.

In the edit function we need to go to the `/edit/:id` URI. In Angular this is done through the `$location` service, the `url` method allows you to set the part after the # part of the URI.

The `delete` function cannot be called `delete` since this is a protected keyword, we therefore call it `delete_`. In this fake controller we go to the home page if the delete is confirmed.

Our dummy controller could look like:

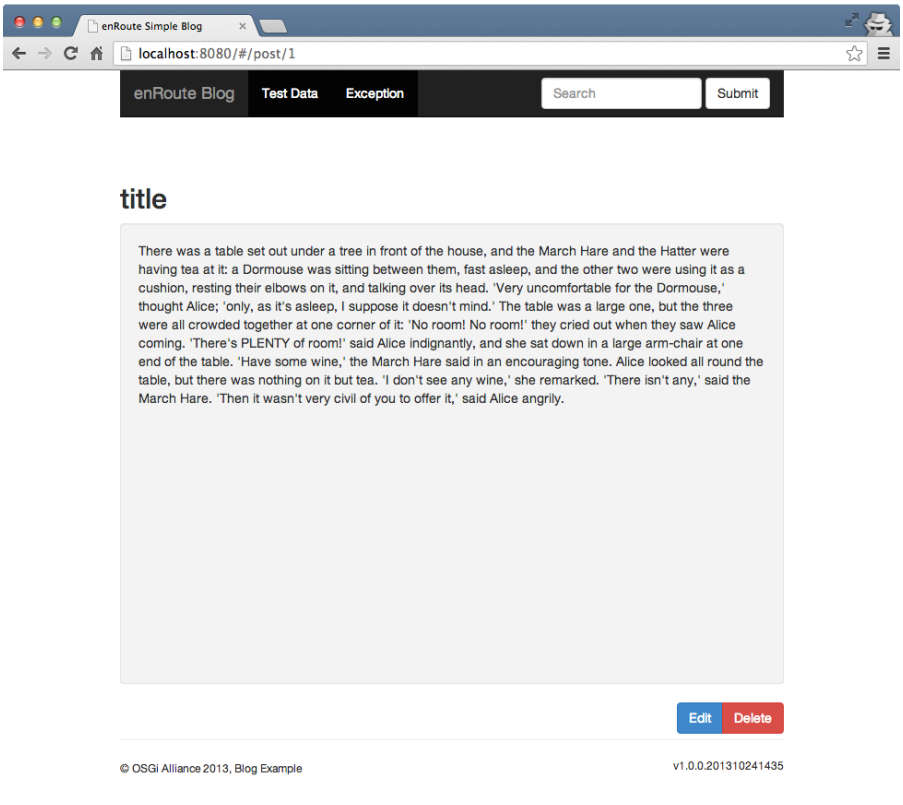
```
function PostView($scope, $routeParams, $location) {
  $scope.post = EXAMPLE;

  $scope.edit = function() {
    $location.url("/edit/" + $routeParams.id);
  }

  $scope.delete_ = function() {
    if (confirm("You are sure you want to delete this post?"))
      $location.url("/");
  }
}
```

In the HTML fragment for the `/post/:id` URI we have two variations. The top `div` tag just shows the graphic elements, the second div tag is show when there are no elements. The `ng-show` attribute is used to show/hide parts of the page based on conditions. This attribute is an Angular expression, if it resolves to a true value (or is undefined) then the element is visible, otherwise it is hidden. In our case, if the `post` variable has no `id` property then we assume there is no post.

The HTML for the `enroute/blog/htm/post/view.htm` template then is:

```
<div ng-show=post.id>
  <h2>{{post.title}}</h2>
  <div style="min-height: 500px" class=well>{{post.content}}</div>
  <div class='btn-group  pull-right'>
    <button class='btn btn-primary' ng-click=edit()>Edit</button>
    <button class='btn btn-danger' ng-click=delete_()>Delete</button>
  </div>
</div>
<div ng-show=!post.id>No such post</div>
```

And this should look like this on http://localhost:8080/#/post/1:



If you're wondering why I am using private mode, well, this makes testing more consistent since a lot state is not kept between sessions.

Blog Post Editing

We need to edit new posts as well as existing post. Obviously this can share a lot of code. The best way in these cases is to use two routes to two different controllers but share the HTML fragment. The common behavior can then be shared in a module function. Lets first add the new routes:

```
$routeProvider.when('/edit', {
  templateUrl : '/enroute/blog/htm/post/edit.htm',
  controller : PostNew
});
$routeProvider.when('/edit/:id', {
  templateUrl : '/enroute/blog/htm/post/edit.htm',
  controller : PostEdit
});
```

For an existing Blog Post we have the following functions:

- initial value must be the existing Blog Post identifed by the `$routeParams.id`
- `save` — Save the Blog Post, and then view the page (go to `/post/:id`)
- `cancel` — Just ignore the changes and go to the home page
- `reset` – Reset the page to the original value.
- `isPristine` — Detect that no edits have been made (used to enable/disable the buttons).

For a new Blog Post, we have the following functions:

- initial value must be empty
- `save` — Create the Blog Post, and then view the page (go to `/post/:id`)
- `cancel` — Just ignore the changes and go to the home page
- `reset` – Reset the page to empty
- `isPristine` — Detect that no edits have been made (used to enable/disable the buttons).

We can capture the common interest in a module private function:

```
function editPost($scope, $location, value) {
  $scope.save = function() {
    $location.url("/post/" + 1);
  }
  $scope.reset = function() {
    $scope.edit = angular.copy(value);
  }
  $scope.cancel = function() {
    $location.url("/");
  }
  $scope.isPristine = function() {
    return angular.equals($scope.edit, value);
  }

  $scope.reset();
}
```

We can now create two controllers, one for new posts and one for existing posts:

```
function PostEdit($scope, $routeParams, $location) {
  editPost($scope, $location, EXAMPLE);
}
function PostNew($scope, $location) {
  editPost($scope, $location, {
  content : "",
  title : ""
  });
}
```

For now, ignore the additional services.

This leaves us with the HTML in `static/enroute/blog/post/edit.htm`. Since we need to disable some of the buttons when they cannot be used (e.g. the Save button should be disabled when you've made no edits), we need to understand the `ng-disabled` attribute. This attribute contains an Angular expression that if it is truthy will disable the corresponding tag.

```
<form>
  <fieldset>
    <input type=text class=form-control ng-model=edit.title>
    <br/>
    <textarea class=form-control style="min-height: 500px;"
      ng-model=edit.content></textarea>
  </fieldset>
  <br/>
  <div class='btn-group pull-right'>
    <button class='btn btn-primary' ng-disabled='isPristine()'
       ng-click=save()>Save</button>
     <button class='btn' ng-click=reset()
       ng-disabled=isPristine()>Reset</button>
     <button class='btn' ng-click=cancel()>Cancel</button>
  </div>
</form>
```

Take a look at http://localhost:8080/#/edit/1 (make sure you refresh).

Blog Post Searching

The last URI we need to support is `/search`. This URI should show a list of Blog Posts that match a globbing expression in the query URI parameter. That is, `/search?query=Tears` should find any blog that has the word Tears in it. Parameters are not explicitly defined in the route.

Adding a new route is simple:

```
$routeProvider.when('/search', {
  templateUrl : '/enroute/blog/htm/post/search.htm',
  controller : PostSearch
});
```

Since we're just creating the user interface we can provide a dummy list for the `posts` variable.

```
function PostSearch($scope) {
  $scope.posts = [EXAMPLE];
}
```

On the HTML page we want to show the query term in an `input` tag so the user can change it. However, we also have a search `input` tag in the navigation bar. It would be nice if these values were synchronized. In Angular this means they have to share a variable in some scope. In this case, the `enBlog` controller provides the top level scope, it should contain the `query` variable so it can be shared between the `PostSearch` controller and the navigation bar. However, making this a simple string object will not suffice ...

The reason is profound. When a variable is referred in Angular it is looked up in its scope. When it is not found it will go to the parent scope, etc. So declaring the `query` variable in the `enBlog`'s `$scope` would work for finding it. However, when the value is assigned it is assigned in the *nearest* scope. So if the `PostSearch` controller has a `ng-model` of `query` then any change in the `input` tag on the search page would set the query variable in the `PostSearch`'s `$scope`, this change would then not be visible in the navigation bar. Sigh.

The solution is to use an object in the `enBlog`'s scope and use a property of that object to contain the `query` variable. Angular will then first lookup this object, and then assign the property on that object. That is, this object would always remain on the same scope.

Since we already need a search function that we can call from the navigation bar as well as the search page we can create a search function in the `enBlog` controller and define the `query` variable as a property of the function object ... Disgusting for a Java developer, but daily business for a Javascript developer.

So let us adjust the `enBlog` controller to define the global `search` function.

```
  command : '@command'
});

$scope.search = function(query) {
  $location.url("/search");
  $location.search({query : query});
}
```

Since the `search` function is now defined in the `enBlog` module, we can adapt the navigation bar in the `index.html` file:

```
<form class="navbar-form navbar-right" role="search">
  <div class="form-group">
    <input type="text" class="form-control"
      ng-model=search.query placeholder="Search">
  </div>
  <button type="submit" ng-click=search(search.query)
    class="btn btn-default">Submit</button>
</form>
```

Next stop is to show an input field for a search term and then our posts with the `ng-repeat` attribute as we've seen before. This is in the `static/enroute/blog/htm/post/search.htm` file.

```
<form>
  <fieldset>
    <legend>Search Posts</legend>
    <div class=input-append>
      <input type="text" ng-model=search.query
        placeholder="Search in a blog">
      <button type="submit" class="btn"
        ng-click=search(search.query)>Search</button>
    </div>
  </fieldset>
</form>
<table class='table table-striped table-condensed' ng-show=posts.length>
  <colgroup><col style="width: 20%"><col style="width: 80%"></colgroup>
  <thead>
    <tr>
      <th>Time</th>
      <th>Post</th>
    </tr>
  </thead>
  <tbody>
    <tr ng-repeat='post in posts'>
      <td>{{post.created | date}}</span></td>
      <td><p><b>{{post.title}}</b></p>{{post.content}}</td>
    </tr>
  </tbody>
</table>
```

We can take a look at the page here: http://localhost:8080/#/search?query=Tears. Notice how you can type text in the navigation field as well as in the input field on the page.

Linking

We should be able to click on a post in the list of Blog Posts and view that post. This could be implemented with a link (the `a` tag). The best solution however is to use a function `go`. Since this function is quite useful in many places we add it in the `enBlog` outer controller so that it is available to all HTML, in any fragment.

```
$scope.go = function(url) {
  $location.url(url);
}
```

We can now change the table to let the Blog Post line link to viewing the Blog Post in the `static/enroute/blog/htm/post/search.htm` file.

```
<tbody>
  <tr ng-repeat='post in posts' ng-click='go("/post/"+post.id)'>
    <td>{{post.created | date}}</span></td>
```

Truncate

In the table that displays the Blog Posts we show the complete content of the blog. This looks awkward if the content is very long, ideally we would like to limit this to 100-200 characters. Earlier we discussed the filters in Angular. These are functions that can process a value to limit it, order it, or format it. Truncating text sounds like a typical use case. The HTML in `static/enroute/blog/htm/post/search.htm` can be adjusted like:

```
    <td>{{post.created | date}}</span></td>
    <td><p><b>{{post.title}}</b></p>{{post.content | truncate: 200 }}</td>
  </tr>
```

However, there is the unfortunate fact that Angular does not have a `truncate` filter. Fortunately, it is possible to add a new filter by declaring it your module. A module has a method to register a filter factory function.

This function is called whenever a filter is needed, it should return a function that takes the following arguments:

- the input (the string to truncate, or more general, whatever is the result of the expression before the pipe operator).
- the parameter of the filter, in our case the truncate length.

This should do the work:

```
MODULE.filter('truncate', function() {
  return function(input, l) {
    if (!input)
      return '';
    else if (l > input.length)
      return input;
    else
      return input.substring(0, l);
  }
});
```

Check it out: http://localhost:8080/#/search?query=Tears

Sorting

The table of Blog Posts is currently unordered, we show it in the order we get from the system. We can sort the elements in the `posts` variable while we are running the `ng-repeat`, the `orderBy` filter does this trick. It takes the name of the column to sort on as argument. We can assign a variable in the current scope to hold this column, lets call it `column`.

```
<tbody>
  <tr ng-repeat='post in posts | orderBy: (column || "modified")'
    ng-click='go("/post/"+post.id)'>
```

It is generally nice if we can click on the column heading to sort on that column. We can use an `ng-click` attribute on the column head to set this variable to the right column.

```
<th ng-click="column='modified'">Time</th>
<th ng-click="column='title'">Post</th>
```

Check it out: http://localhost:8080/#/search?query=Tears

Feedback Which Columns is Sorted

This will do the ordering but it will not provide any feedback which column is ordered. The following changes tot the `static/enroute/blog/htm/post/search.htm` file will make this more clear.

```
<tr>
  <th ng-click="column='modified'">Time
    <span ng-show="column=='modified' || column==undefined">▽</span>
  </th>
  <th ng-click="column='title'">Post
    <span ng-show="column=='title'">▽</span>
  </th>
</tr>
```

Check it out: http://localhost:8080/#/search?query=Tears.[3]

Home

So far the home page still shows "Hello World". It would be nice to show all the blogs on this page. Displaying this list is of course identical to displaying the table on the search page. It would be a waste to copy that table because it is likely we will add more features to this table in the future. How can we share this view of the `posts` variable?

The trick in Angular is to use the `ng-include` tag. This tag has a `src` attribute that is an Angular expression that results in the path of the template. It is therefore possible to put the table we used in the search page in a separate file: `/enroute/blog/post/table.htm` page.

[3] As a side note, this feature would probably work very well as an Angular directive. Directives can give HTML tags, attributes, or classes new behavior. They are not handled in this tutorial.

```
<table class='table table-striped table-condensed' ng-show=posts.length>
  <colgroup><col style="width: 20%"><col style="width: 80%"></colgroup>
  <thead>
    <tr>
      <th ng-click="column='modified'">Time
        <span ng-show="column=='modified' || column==undefined">▽</span>
      </th>
      <th ng-click="column='title'">Post
        <span ng-show="column=='title'">▽</span>
      </th>
    </tr>
  </thead>
  <tbody>
    <tr ng-repeat='post in posts | orderBy: (column || "modified")'
        ng-click='go("/post/"+post.id)'>
      <td><span>{{post.created | date}}</span></td>
      <td><p><b>{{post.title}}</b></p>{{post.content | truncate: 200}}</td>
    </tr>
  </tbody>
</table>
```

We can then include this file in the `/enroute/blog/home.htm`

```
<ng-include src="'/enroute/blog/htm/post/table.htm'"></ng-include>
```

And `/enroute/blog/post/search.htm` page:

```
    </fieldset>
</form>
<div ng-show="posts.length &gt 0">
  <ng-include src="'/enroute/blog/htm/post/table.htm'"></ng-include>
</div>
```

To make this work, the Home controller must set the post variable:

```
function Home($scope) {
  $scope.posts = [EXAMPLE];
}
```

Enjoy at : http://localhost:8080/#/ and verify the search page at http://localhost:8080/#/search?query=Tears.

Test Data

In the previous step we used test data to provide us with some data to play with and just set the `post` variable. However, we no longer have that luxury. What would happen when we showed the search page, or the home page? Instead of setting a variable, we need to reload the page to take advantage of any changes in the server. We should therefore change the `TESTDATA` command in the enBlog controller as follows:

```
$scope.testdata = function() {
  Command.get({command : 'TESTDATA'},
    function() {
      $route.reload();
       ok();
     },
     error);
}
```

The use of this function will become more clear later.

References

- Angular Routing
  http://docs.angularjs.org/tutorial/step_07
- Cucumber
  http://cukes.info/

8. Blog Backend

Goal

Understand why you want a facade in the OSGi back-end.

Prerequisites

First ensure that you have checked out the following branch in the workspace.

```
$ git checkout 06-blog-gui
```

You should compare the the project against your own solution of the previous step. you can also continue with your previous step. If you continued from the previous phase then the framework should still be running. Verify this; if it is not running then start it. We will update the framework incrementally.

Background

OSGi provides the foundation on which bndtools supports web application development. In the previous steps we've shown how to use REST to communicate to the backend and then developed a user interface base don Angular. In this chapter we provide a simple *facade* for the Javascript code.

A facade is required because normal OSGi services are not applicable to use from another process. It is a fundamental aspect of web design that an endpoint is an access point from the outside world and vulnerable to attacks. Writing plain OSGi services in a way that they can be access from an untrusted location would make them very complicated to write and a system would be very hard to verify since there would be so many points of access.

There is even another reason. In practice, it is often necessary to use services in a specific way that requires knowledge of the current user. If all OSGi services had to be aware of this then they would become intricately tied to the authorization and user management subsystems. It is better to separate these concerns. The responsibilities of the facade are therefore:

- Authorize the request
- Parametrize any user aspects
- Combine the results of one or more OSGi services

A facade can become quite large because it provides the endpoint for a potentially large Javascript application. To keep things manageable, a facade should never do real work, it should authorize, verify, and call other services for the low level grunt work.

Blog App Facade

In a previous step we created a Blog App component that implemented the `getCommand` method (the `GET /rest/command` endpoint). We now have to extend it for the user interface we've created in the previous step. This user interface uses a Blogpost resource that requires the following URIs:

```
POST   /rest/blogpost                  BlogPost         create
GET    /rest/blogpost/<id>             BlogPost         read
GET    /rest/blogpost?query=<query>    [ BlogPost ... ] query
POST   /rest/blogpost/<id>             BlogPost         update
DELETE /rest/blogpost/<id>                              delete
```

Reading

In this step we will read a post from the server. We need to add a resource factory to the Javascript code in `blog.js` for our Blog Posts. We call this resource factory `Post`. As explained for the `Command` resource factory, we need to declare this as a module variable:

```
var MODULE = angular.module("Blog", [ "ngResource" ]);
var Post, Command, ok, error;
```

In the `enBlog` controller we then create the `Post` resource factory. The factory creation can specify default values for the URI. These default values can be specified in the invocation, from the underlying resource object, or from the defaults. If the value is a string that starts with an @ character than the underlying object's property is used to retrieve the property with that name. For example, `xyz:"@id"` indicates that the id property of the resource object is used as the value for the URI `xyz` parameter. URI parameters end up either in the URI's path or as query parameter if the resource URI does not specify that property. Therefore change the `enBlog` controller:

```
ok = function(result) { $scope.message = ""; }
Post = $resource("/rest/blogpost/:id", {id : '@id'});
```

We view the Blog Post from the `/post/<id>` route, which ends up in the `PostView` controller so we must adapt this controller to fetch the content from the server.

We the `$routeParams` service since contains the `id` property, which we must specific. While we're at it, we can also call the `$remove` function on the object if the `Delete` button is pressed.

So change the `PostView` controller to:

```
function PostView($scope, $routeParams, $location) {
  $scope.post = Post.get($routeParams,ok,error);

  $scope.edit = function() {
    $location.url("/edit/" + $routeParams.id);
  }

  $scope.delete_ = function() {
    if ( confirm("You are sure you want to delete this post?") )
      $scope.post.$remove(function(d) {
        $location.url("/");
        ok();
      }, error);
  }
}
```

To test this we need to add the `GET /rest/blogpost/<id>` endpoint to the `BlogApp` class. In this step we will store the Blog Posts in a Map, i.e. an in-memory 'database'.

We will need some fields to hold our blogs (a `Map`) and to assign unique ids we need an `AtomicLong`, we can use its getAndIncrement to create a unique id. For nicer formatting, it is best to start the first id at a nice large number so most blog post ids have the same number of digits.

```
public class BlogApp implements ResourceManager {
  final Map<Long, BlogPost>    posts= new ConcurrentHashMap<Long, BlogPost>();
  final AtomicLong ids         = new AtomicLong(1000);
```

In the previous incarnation we had a static public class `BlogPost`. Let's first move this out into its own class in the same package. This keeps the facade cleaner. While we're at it, add some fields to give it an id and a last modified time. We will not need a constructor anymore.

```
public class BlogPost {
  public long           id;
  public String         title;
  public String         content;
  public long           created;
  public long           lastModified;
}
```

Removing the constructor has created an error in our `TESTDATA` command. For testing it is always nice to have some test data. So let us change the `TESTDATA` command we added earlier to add some records. For brevity, we do this in a separate method:

```
switch (command) {
case TESTDATA :
  testdata();
  return null;
case EXCEPTION :
```

The `testdata` method must add some methods to the posts map. For brevity the length of the `content` and the number of Blog Posts is limited. For testing purposes it is better if the `content` is much longer and there are more Blog Posts.

```
void testdata() throws Exception {
    post("Down the Rabbit Hole", "Alice was beginning ...");
    post("The Pool of Tears", "`Curiouser and curiouser!' ...");
    ...
}
BlogPost post(String title, String content ) {
  BlogPost p = new BlogPost();
  p.title = title;
  p.content = content;
  p.id = ids.getAndIncrement();
  p.lastModified = p.created = System.currentTimeMillis();
  posts.put(p.id, p);
  return p;
}
```

We now have some content! Let's read it! The following method is available under the `GET /rest/blogpost/<id>` URI.

```
public BlogPost getBlogpost(Options opts, long id) throws Exception {
  return posts.get(id);
}
```

Make sure it is all saved and go to http://localhost:8080/#/post/1000. Since we have not activated the test data yet, we will see that there are no posts. Click on the Test Data button in the navigation bar and see the display reload with the first blog post. See that the `testdata` function in `blog.js` is reloading the page after the method returns.

Delete

The Blog Post view pane also has a `Delete` button so this is our next candidate. The delete REST URI is `DELETE /rest/blogpost/<id>`, the corresponding method is then:

```
public void deleteBlogpost(Options opts, long id) throws Exception {
  posts.remove(id);
}
```

See if you can delete the post http://localhost:8080/#/post/1000.

List

The list function is used from the home page (`/`)and from the search page (`/search?query=`). We therefore need to update the `Home` controller:

```
function Home($scope) {
  $scope.posts = Post.query(ok, error);
}
```

And the `PostSearch` controller:

```
function PostSearch($scope, $routeParams) {
  $scope.posts = Post.query($routeParams, ok, error);
}
```

Listing the Blog Posts is the `GET /rest/blogpost?query=<query>` URI. It is a bit more complicated since we need to search the Blog Posts, the `query` parameter specifies a globbing expression (wildcards). To keep things relatively simple we turn the query into a regular expression and search this pattern then in the `title` and the `content`.

To get access to the query parameter we need to extend the Options interface and add a `query()` method:

```
interface BlogPostOptions extends Options {
  String query();
}
```

We can then write the `getBlogPost` method, using the interface as the first's argument's type.

```
public Iterable<BlogPost> getBlogpost(BlogPostOptions options)
  throws Exception {
  String query = options.query();
  if (query == null) { return posts.values(); }

  query = query.trim();
  query = query.replace("*", ".*").replace("?", ".?");
  query = "(" + query.replaceAll("[\\s+]+ ", ")|(") + ")";
  Pattern q = Pattern.compile(query, Pattern.CASE_INSENSITIVE);
  List<BlogPost> result = new ArrayList<BlogPost>();

  for (BlogPost p : posts.values()) {
    if (q.matcher(p.content).find() || q.matcher(p.title).find())
      result.add(p);
  }
  return result;
}
```

Save and go to http://localhost:8080/#/search?query=Tears. If you do not see anything, you might try to create new `Test Data`. You can see the full list you created in the `testdata` method here http://localhost:8080/#/

Create

We create a new Blog Post from the `/edit` route. This route is managed by the `PostNew` controller. We should adapt this controller as follows:

```
function PostNew($scope, $location) {
  editPost($scope, $location, new Post({
     content : "",
     title : ""
  }));
}
```

The `editPost` module function must save the object. Since we create a `Post` resource, we can call the `$save` method on it. This method is automatically provided by the resource factory. This is the new `save` function in the `editPost` function:

```
$scope.save = function() {
  $scope.edit.$save(function(post) {
    $location.url("/post/" + post.id);
    ok();
  }, error);
}
```

To create we need to have the content. The user interface sends us a JSON message containing the `title` and the `content` of the Blog Post in the body of the web request. Since we prefer not to muddle with JSON decoding etc. it would be nice if we could access the Blog Post in a type safe way. We can!

The REST server provides access to the body of the request through the Options interface. However, to decode the JSON stream it needs the actual type it should be parsed into. This can be a Map, but it is nice to get a real object. Since the user interface sends us it in the format of a Blog Post we can use the `BlogPost` class. The REST server finds the desired type by looking at the _() method on the Option's argument. We therefore need to add this method to our `BlogPostOptions` interface.

```
interface BlogPostOptions extends Options {
  String query();
  BlogPost _();
}
```

The create URI is `POST /rest/blogpost`, with a body of the Blog Post.This maps to the following method:

```
public BlogPost postBlogpost(BlogPostOptions opts) throws Exception {
  BlogPost post = opts._();
  post.id = ids.incrementAndGet();
  post.created = System.currentTimeMillis();
  post.lastModified = System.currentTimeMillis();
  posts.put(post.id, post);
  return post;
}
```

Time to test this http://localhost:8080/#/edit

Update

Updates are already handled in the `editPost` function. However, we need to adapt the controller for the `/edit/:id` route, the PostEdit controller. This controller must get the resource as a resource. However, what is returned is a place holder so we cannot call `editPost` yet. In the ok callback we then call `editPost`.

```
function PostEdit($scope, $routeParams, $location) {
  var post = Post.get($routeParams, function() {
    editPost($scope, $location, post);
    ok();
  }, error);
}
```

The following method is available under the `POST /rest/blogpost/<id>` URI:

```
public BlogPost postBlogpost(BlogPostOptions opts, long id) throws Exception{
  BlogPost post = opts._();
  BlogPost old = posts.get(id);
  old.content = post.content;
  old.title = post.title;
  post.lastModified = System.currentTimeMillis();
  return old;
}
```

Take a look at http://localhost:8080/#/edit/1002. Don't forget to refresh and create test data.

9. Service Based Design

Goal

Understand proper OSGi service based design and why the previous step was bad design.

Prerequisites

First ensure that you have checked out the following branch in the workspace.

```
$ git checkout 07-blog-facade
```

You should compare the the project against your own solution of the previous step. you can also continue with your previous step. If you continued from the previous phase then the framework should still be running. Verify this; if it is not running then start it. We will update the framework incrementally.

Background

OSGi is *not* an application server, it is a way to build complex applications from components. A primary aspect of this is that OSGi is a *service* based framework.Services provide an elegant way for components to collaborate without getting bogged down in each other's details. They fulfill this role by sharing an object that has a well defined contract.

A very important part of Object Oriented design is cohesion. It is a measure of how related the constituents of a group/module are. Uncohesive designs have a tendency to be hard to maintain because they need to be changed, affecting constituents that have no relation whatsoever with the requested change.

In the previous step we created a facade and the implementation code of the desired functionality (a Blog Manager) into one class. The primary purpose of the facade is to be the entry point of the system and handle authentication, authorization, and user related issues. In real applications this is a considerable effort.

From an OSGi perspective, the `BlogApp` class should depend on a `BlogManager` service that would implement the desired functionality.

In this step we will split the current facade in three projects:

- `osgi.enroute.blog.api` — An API project. This is in general required if the API can have multiple providers.
- `osgi.enroute.blog.memory.provider` — A memory based implementation of the API
- `osgi.enroute.blog.appl` — The current application project with the facade.

Create an API

We will first create a new project that will contain the API of a Blog Manager service. This project is called `osgi.enroute.blog.api`. Make this an empty project, no dependencies.

We need an interface for the Blog Manager service and a BlogPost domain object to hold the data. The Blog Manager API looks as follows:

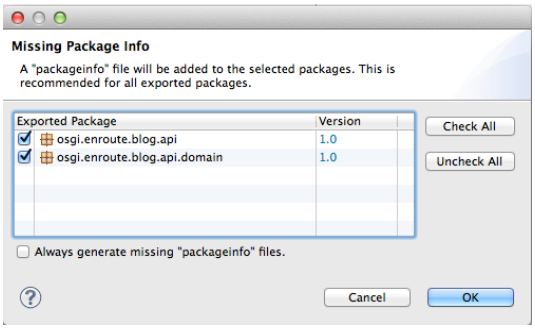
```
package osgi.enroute.blog.api;
import osgi.enroute.blog.api.domain.BlogPost;
public interface BlogManager {
  long createPost(BlogPost post) throws Exception;
  void updatePost(BlogPost post) throws Exception;
  BlogPost getPost(long id) throws Exception;
  Iterable<BlogPost> queryBlogs(String query) throws Exception;
  void deletePost(long id) throws Exception;
}
```

There is likely some controversy around the use of Exception. Checked exceptions are an utterly failed experiment. The only useful exceptions to catch in normal flow code are not exceptions by nature (FileNotFondException for example). Read Bertrand Meyer for more background.

The Blog Post class is placed in a domain package since in practice if often finds use outside the Blog Manager:

```
package osgi.enroute.blog.api.domain;
public class BlogPost {
  public long           id;
  public String         title;
  public String         content;
  public long           created;
  public long           lastModified;
}
```

We must fill in the `bnd.bnd` file to make this into a bundle. Double click the `bnd.bnd` file and select the `Contents` tab. You an now drag the packages with the new classes to the `Exported Packages` list. Since we are exporting this package, bndtools will ask if it should create a `packageinfo` file. Such a file contains the version of the package.



Version are best practice so we select this. This will create two packageinfo files, one in each package. They contain:

```
version 1.0
```

bnd uses this version to create the versions on Export Package statements and can use it to create import ranges in an Import Package statement.

It is very good practice to always add a version to a project with a timestamp as well as a description. After you've done this, the `Source` tab of the `bnd.bnd` file should like something like:

```
Bundle-Version:          1.0.0.${tstamp}
Bundle-Description:      \
 The API for a Blog Manager. A Blog Manager provides create/update/\
 read/list/delete methods for BlogPosts. This is a very simple \
 BlogManager, intended to be used for a single person.

Export-Package: osgi.enroute.blog.api,\
    osgi.enroute.blog.api.domain
```

You can inspect the bundle by double clicking on the JAR file in the `generated` folder.

package-info.java[4]

In the previous section we added a `packageinfo` file. This method has been supported by bnd's predecessor btool since 1999. However, since Java 5 it is also possible to specify annotations on packages. bnd therefore also supports this with an `@Version` annotation. The OSGi Alliance is in the process of standardizing this way of marking the version of a package.

It is therefore possible to replace the `packageinfo` file with a `package-info.java` file. This is a standard Java file recognized by the compiler. This file then has to look like:

```
@aQute.bnd.annotation.Version("1.0.0")
package osgi.enroute.blog.api;
```

This will require the annotations to be on the build path. You can find these in the JPM repository, drag the revision of `biz.aQute.bnd.annotations` to the `Build` tab's `Build Path` list panel. Or alternatively add it to the `Source` tab:

```
-buildpath: biz.aQute.bnd.annotation
```

Provider Types[5]

In a contract there usually is an asymmetry in roles. One side is usually the primary *provider* of the obligations in the contract and the other is the primary *consumer* of those obligations. For example, for the OSGi Event Admin service the implementation of the Event Admin interface is clearly the provider of this contract. Bundles that post and send events are the consumers.

In general consumers enjoy wide backward compatibility. That is, new versions of the contract are designed so that an existing consumer gets the same guarantees. For example, a future version of the Event Admin specification could add additional methods or guarantees. Since older consumers have no knowledge of those new features they cannot be broken by them as long as the existing methods behave like before. There is no luxury for providers.

Since OSGi uses semantic versioning we need to know if an interface is supposed to be implemented by a provider by a consumer since this affects the version ranges generated by bnd. For example, adding a method to a provider interface is a minor change while a change to a consumer type is a major change.

There are a number of bnd annotations[6] for this purpose.

- `@ProviderType` — Marks a type as being implemented by the provider
- `@ConsumerType` — Marks a type as being implemented by a consumer

It is very good practice to mark types in specifications. In this case the `BlogMamager` class must be marked as a `@ProviderType`.

```
import osgi.enroute.blog.api.domain.BlogPost;
import aQute.bnd.annotation.ProviderType;
@ProviderType
public interface BlogManager {
```

The Memory Provider

Just for testing purposes it is nice to have a very simple implementation of the API. Since we already implemented virtually all the API's method in the facade we can easily create a new project and copy the implementation from BlogApp to this project.

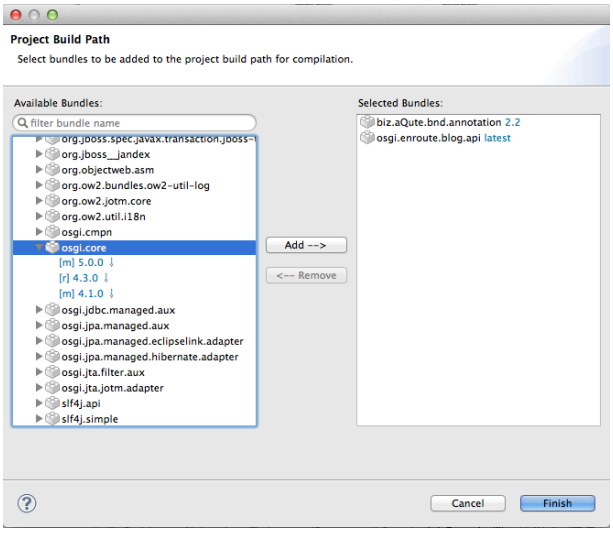
So create a new, empty, bndtools project `osgi.enroute.blog.memory.provider`. The memory provider has a dependency on the API project and we will use the bnd annotations. We will also need the OSGi core. They must therefore be added to the project. You can double click the `bnd.bnd` file and select the `Build` tab. You can click on the + on the right top of the `Build Path` list panel. You then get a dialog that allows you to find dependencies and add them to the build path.

---

[4] You can skip this step if you want, it is quite advanced and does not affect the functionality

[5] This step can be skipped. It does not involve use functionality

[6] The Alliance is in the process of standardizing them, though they will go into another package



Alternatively you can add this in the `Source` tab:

```
-buildpath: osgi.core,\
    biz.aQute.bnd.annotation,\
    osgi.enroute.blog.api;version=latest
```

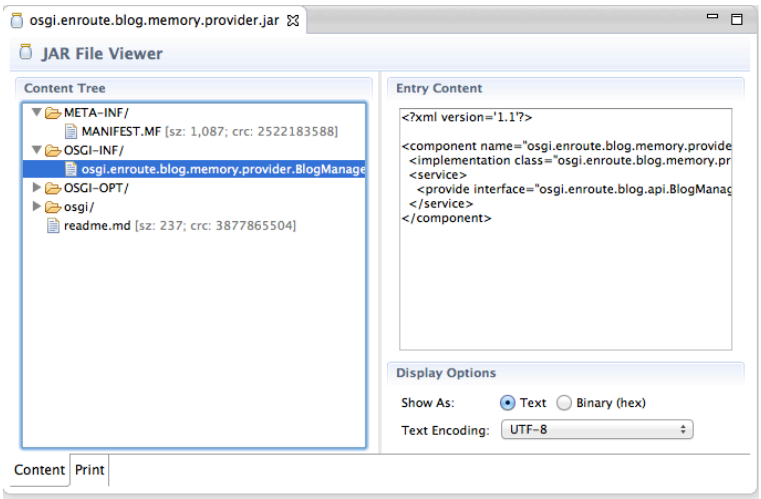
Notice that the `version=latest` is used to specify a dependency on another project in the same workspace. The other dependencies have no minimum version, this means they will use the lowest version in the repositories to compile against.

Then add an `osgi.enroute.blog.memory.provider.BlogManagerImpl` class. This class is a *component*. In OSGi speak, a Declarative Services (DS) component. These components are very well supported by bnd. Though DS requires a yucky XML file, bnd can generate this XML from a few annotations and the available type information.The skeleton of a component is:

```
@Component
public class BlogManagerImpl implements BlogManager {}
```

In this workspace, the `cnf/build.bnd` file has a `Service-Component` header set to `*`, this gives bnd the permission to find any component that is placed in the bundle and automatically generate the XML files for them.

You can look at what bnd does with this annotation by double clicking the `generated/osgi.enroute.blog.memory.provider.jar` file.



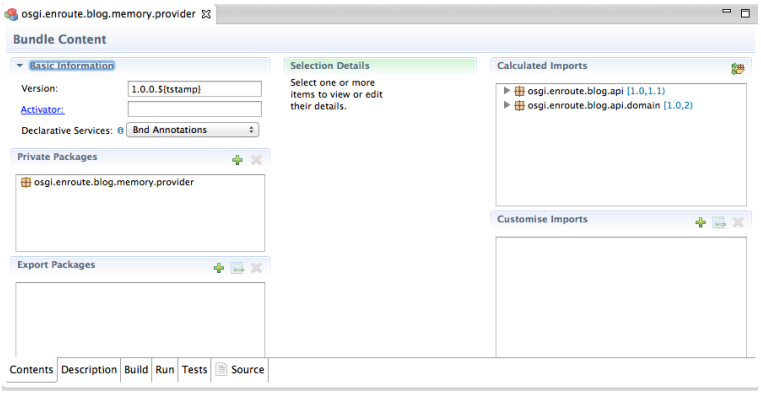
Note that the implementation is almost fully contained in the `BlogApp.java` file. You can try to write this code yourself, copy it from the `BlogApp.java`, or copy it from here:

```
@Component
public class BlogManagerImpl implements BlogManager {
  final Map<Long, BlogPost>          posts  =
    new ConcurrentHashMap<Long, BlogPost>();
  final AtomicLong                   ids = new AtomicLong(1000);

  @Override
  public long createPost(BlogPost post) throws Exception {
    BlogPost newPost = new BlogPost();
    newPost.id = ids.incrementAndGet();
    newPost.created = System.currentTimeMillis();
    posts.put(newPost.id, newPost);
    update(post, newPost);
    return newPost.id;
  }
  private void update(BlogPost original, BlogPost toBeUpdate) {
    toBeUpdate.content = original.content;
    toBeUpdate.title = original.title;
    toBeUpdate.lastModified = System.currentTimeMillis();
  }
  @Override
  public void updatePost(BlogPost post) throws Exception {
    BlogPost p = posts.get(post.id);
    if (p == null)
      throw new FileNotFoundException("No such post " + post.id);
    update(post, p);
  }
  @Override
  public BlogPost getPost(long id) throws Exception {
    return posts.get(id);
  }
  @Override
  public Iterable<BlogPost> queryBlogs(String query)
      throws Exception {
    if (query == null) {
      return posts.values();
    }
    query = query.trim();
    query = query.replace("*", ".*").replace("?", ".?");
    query = "(" + query.replaceAll("[\\s+]+ ", ")|(") + ")";
    Pattern q = Pattern.compile(query, Pattern.CASE_INSENSITIVE);
    List<BlogPost> result = new ArrayList<BlogPost>();
    for (BlogPost p : posts.values()) {
      if (q.matcher(p.content).find() || q.matcher(p.title).find())
        result.add(p);
    }
    return result;
  }
  @Override
  public void deletePost(long id) throws Exception {
    posts.remove(id);
  }
}
```

Bundle Design

Currently we have not defined the content of this bundle yet. Clearly we need to include the `osgi.enroute.blog.memory.provider` package as a Private package. We can select the `bnd.bnd's Contents` tab and then dragging the package to the `Private Package` list panel.

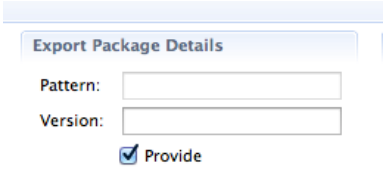
This will create an import for the API. This can be seen on the Contents tab, right side, the Calculated Imports tab. If they do not show up, close the pane and open it again.



This is not a good practice for providers. A provider is tightly coupled to the API it provides, there is very little benefit in having another bundle carry the API packages and it is awkward to use when people give you bundles that drag in lots of other bundles. In principle, a bundle should be self contained and not require lots of other bundles (depending on other services is ok though). Particularly bad practice is to put many APIs together (like for example the OSGi Compendium) and use them as a run bundle (to compile against is ok), which creates lots of unnecessary constraints.

We should therefore export the packages in this provider bundle. We can achieve this by dragging the packages in the `Calculated Imports` directly to the `Exported Packages` pane. This will create an `Export-Package` clause, which will instruct bnd to automatically include these packages in the bundle.

When you then select the exported packages in the Exported Packages pane you see that in the middle you can customize this pane.



In the `Export Package Details` you see a `Provide` checkbox. The purpose of this checkbox is to tell bnd that it should treat this bundle as the API provider of those packages. This, again, has to do with semantic versioning. You should *not* provide a version, the version is derived from the bundle that contains this package. Alternatively, you can also add this via the `Source` tab.

```
-buildpath: osgi.core,\
    biz.aQute.bnd.annotation,\
    osgi.enroute.blog.api;version=latest

Private-Package: \
    osgi.enroute.blog.memory.provider

Export-Package: osgi.enroute.blog.api;provide:=true,\
    osgi.enroute.blog.api.domain;provide:=true
```

Running

The Memory provider is not running, we run the application from the `osgi.enroute.blog.appl` project. We therefore have to add this bundle to the `-runbundles` instruction in the `bnd.bnd` file in that project. We can either do this via the `osgi.enroute.blog.appl bnd.bnd` editor's Run tab, we can then add a new run bundle with the + on the right corner of the `Run Bundles` list pane. Or we can do it by hand in the `Source` tab:

```
-runbundles: \
  cnf.run.base, \
  cnf.run.web, \
  cnf.run.web.debug, \
\
  aQute.twitter.bootstrap;version=3.0.0, \
  aQute.google.angular.stable;version=1.0.8, \
\
  osgi.enroute.blog.memory.provider;version=latest
```

Make sure you are in the right project! You need to be in the `osgi.enroute.blog.appl` project.

Once this Blog App is done, you can verify if it works by looking at X-Ray http://localhost:8080/system/console/xray. Since we are not using its service yet, it should have registered a white service with a solid outline:

README

Though this implementation bundle is rather useless in the greater scheme of things, in general you should expect these bundles to be used by others. One of the worst things is to encounter a bundle and have no clue what it does or how it can be used. Adding a `readme.md` file can make a tremendous difference. The `md` extension is markdown, markdown is very suitable for this kind of readme files since it looks good without a formatter as well.

So create a `readme.md` file in the `osgi.enroute.blog.memory.provider` project:

```
# enRoute Blog Manager — In Memory Provider
This bundle implements the enRoute Blog example BlogManager API as an in-memory
database.

## Configuration
There is no configuration required.

_${Bundle-SymbolicName}-${Bundle-Version}_
```

You can use any of bnd's macros in this file if you instruct bnd to preprocess it, that is add a clause to the `-includeresource` instruction and surround the path (relative to the `bnd.bnd` file) with curly braces:

```
-includeresource: {readme.md}
```

As a bonus, if you use jpm, it will show up on the readme tab.

Adapting Blog App

Next we must adapt the `osgi.enroute.blog.appl` project. We have to add the dependency to the API project so we can see the interface and domain object. Go to the bnd.bnd editor, select the Build tab, and use the `+` to add the dependency on `osgi.enroute.blog.memory.api` project.

We can now delete the `BlogPost` class since we will get it from the API.

This Blog App now depends on a service delivered by the memory provider. We need a Blog Manager service, this is achieved with an `@Reference` annotation on a setter method. Add this to the `BlogApp` class.

```
@Reference
void setBlog(BlogManager bm) {
  this.blogs = bm;
}
```

This requires the creating of a field: `blogs`. The endpoint methods in the facade now need to be changed to use the `blog` field instead of directly handling the Blog Posts itself.

```
public Iterable<BlogPost> getBlogpost(BlogPostOptions options)
    throws Exception {
  return blogs.queryBlogs(options.query());
}
public BlogPost postBlogpost(BlogPostOptions opts) throws Exception {
  long id = blogs.createPost(opts._());
  return blogs.getPost(id);
}
public BlogPost postBlogpost(BlogPostOptions opts, long id) throws Exception{
  BlogPost post = opts._();
  blogs.updatePost(post);
  return post;
}
public void deleteBlogpost(Options opts, long id) throws Exception {
  blogs.deletePost(id);
}
public BlogPost getBlogpost(Options opts, long id) throws Exception {
  return blogs.getPost(id);
}
```

If you now look in X-Ray http://localhost:8080/system/console/xray you will see that the memory provider and the application bundle are connected through the Blog Manager service.

You an now test the application again at http://localhost:8080/#/

Lazyness

If you play with the application and make some code changes to explore the possibilities you will notice that the memory provider is continuously loosing its contents. Though it is a memory provider, implying there are no guarantees, it is annoying.

The reason is that DS components are by default lazy. Though they register a service, they do not create a service object until the service is actually used. This is great because it improves startup time and conserves resources. However, in our case when we make a change in the application bundle it will get the service, which will create the memory provider component, and then will unget the service because you made a code change that forced an update of the bundle. This unget will cause the memory provider component to be deactivated, loosing its contents.

We can improve on this by marking the component immediate, this will register its service and activate the component immediately.

```
@Component(immediate = true)
public class BlogManagerImpl implements BlogManager {
```

References

- Bertrand Meyer on Exceptions
  http://se.ethz.ch/~meyer/publications/methodology/exceptions.pdf
- Markdown
  http://daringfireball.net/projects/markdown/
- JPM
  https://www.jpm4j.org

10. OSGi Testing

Goal

Learn how to test code inside an OSGi framework, from inside Eclipse and from the command line.

Prerequisites

First ensure that you have checked out the following branch in the workspace.

```
$ git checkout 08-memory-provider
```

You should compare the the project against your own solution of the previous step. you can also continue with your previous step.

If you continued from the previous phase then the framework should still be running. Verify this; if it is not running then start it. We will update the framework incrementally.

Background

The need testing is clear and does not need explanation. JUnit unit tests are the same as normal projects as long as they do not need OSGi. In many cases mocks can be used but at a certain level it is necessary to run the tests in a real framework. bnd(tools) has extensive support for this. It is possible to run JUnit tests with an OSGi framework using the same user interface support as standard JUnit tests.

In this case we need to test the Blog Manager API. Since we cannot test APIs without an implementation we will set up the default environment to test the Memory provider. This allows us to quickly test our testing code. However, we can use the bundles with the tests later in the JPA step.

Creating a Test Project

We create a special project to hold our tests so that we can use in different setups so we can use the test bundle to test different providers. We call this project: `osgi.enroute.blog.test`. Again, make it an empty bndtools project so we add all the information by hand.

We will need the following dependencies on our build path:

- `osgi.core`
- `org.apache.servicemix.bundles.junit` version 4.11 or later
- `aQute.dstest`
- `osgi.enroute.blog.api` (version=latest since this is a project)

You can add these dependencies on the `Build` tab via drag and drop, the +, or via the `Source` tab. We also need to add some bundles to create a test environment. This consists of the following parts:

- `-runfw` —This specifies the Bundle Symbolic Name and version range for the framework we use. This framework is added to the `-runpath`.
- `-runpath` — Specifies the JARs that are on the normal class path of the application. That is, these JARs are not bundles. However, any Export-Package headers are parsed and added to the system packages exported by the framework.
- `-runbundles` — The `-runbundles` specifies the bundles that should be installed and started.
- `-runvm` —Any arguments to the VM like -ea for assertions or the -X options for memory allocation.
- `-runproperties` —Any properties that will be set as System properties in the running Java VM.

Fortunately, most of these instructions have defaults specified in the `cnf/build.bnd` file. We therefore can set the bnd.bnd file to:

```
Bundle-Version:  1.0.0.${tstamp}
Bundle-Description: \
 This bundle is an OSGi test bundle that tests the enRoute Blog Manager API.\
 Projects that provide this API should configure their bnd.bnd file to run \
 this test so that the providers can test their conformance. This bundle \
 requires DSTest to run so this  bundle should be included.

-buildpath: \
    osgi.core;version='[4.3.1,6)',\
    org.apache.servicemix.bundles.junit;version='[4.11.0,5.0.0)', \
    osgi.enroute.blog.api;version=latest

-runbundles: \
    cnf.run.base, \
    aQute.dstest;version=1.1.0, \
    osgi.enroute.blog.memory.provider;version=latest

-runpath: \
    org.apache.servicemix.bundles.junit;version=4.11
```

Writing a test case

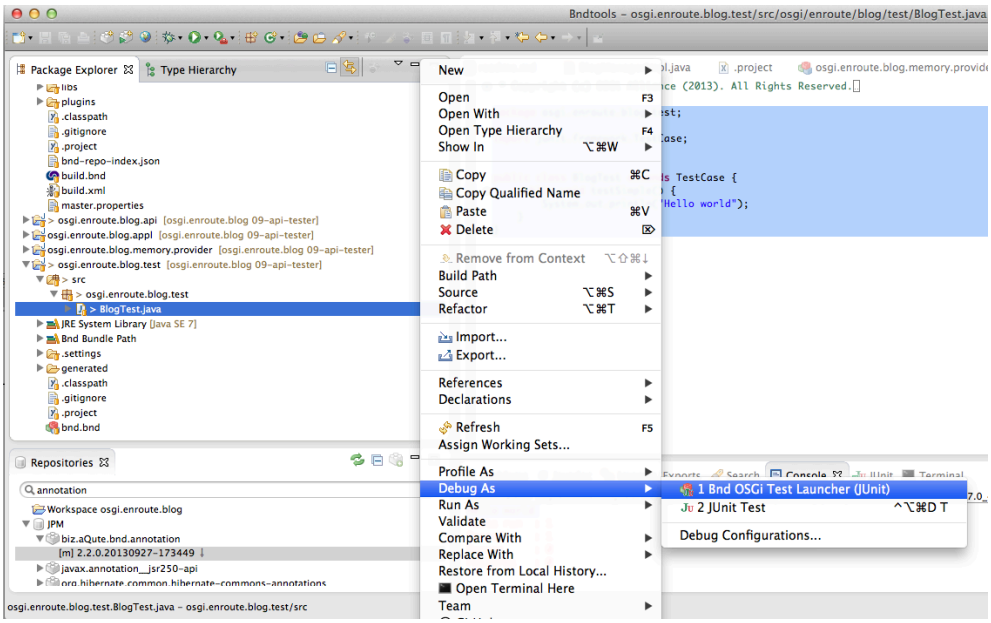
To get started, let's write a minimal test case in the class `osgi.enroute.blog.test.BlogTest`.

```
package osgi.enroute.blog.test;
import junit.framework.TestCase;
import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertThat;

public class BlogTest extends TestCase {
  public void testSimple() {
    System.out.println("Hello world");
  }
}
```

We need to add the test package to the bundle. Add by dragging it to the `Contents` tab's `Private Package` list, or add it through the `Source` tab:

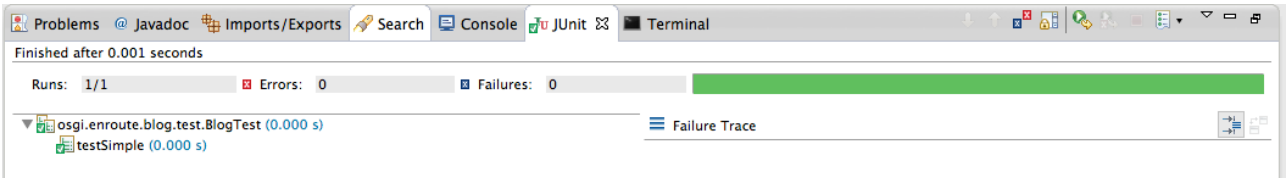
```
Private-Package: \
  osgi.enroute.blog.test

-buildpath: \
```

You can now select the `BlogTest` class, call up the context menu, select `Debug As/Bnd OSGi Test Launcher (JUnit)`:



This will run all the tests in the selected class. You can also select a specific test method in the source fille. JUnit is very well supported in Eclipse, you can trace progress in the JUnit view: the famous green bar we all

love so much (and dread when it turns ugly brown). So after you run your test you can see the results in the JUnit view:



You can also re-run the test, you can debug them, set breakpoints etc. All the functions you're used to are working as the test was a normal JUnit test. However, the current test is not doing anything with the running OSGi framework, or that we need a BundleContext. Though the official way is to create a setter in your test case, there is an easier standard OSGi way, FrameworkUtil. This class can return the Bundle of any class. So let us try this out:

```
public class BlogTest extends TestCase {
  BundleContext context = FrameworkUtil.getBundle(
      BlogTest.class).getBundleContext();
  public void testSimple() {
    System.out.println(Arrays.toString(context.getBundles()));
  }
}
```

Running this gives us the following output in the console:

```
[org.apache.felix.framework [0], org.apache.felix.configadmin [1],
org.apache.felix.log [2], org.apache.felix.scr [3], slf4j.simple [4], slf4j.api [5],
aQute.configurer [6], aQute.executor [7], org.apache.felix.gogo.runtime [8],
org.apache.felix.gogo.shell [9], org.apache.felix.gogo.command [10],
aQute.logger.intrf [11], aQute.dstest [12], osgi.enroute.blog.memory.provider [13],
osgi.enroute.blog.test [14]]
Tests run   : 1
Passed      : 1
Errors      : 0
Failures    : 0
```

As you can see, the end of the console also shows a short summary of the run tests.

Dependencies

In test code we can not rely on DS since it is the test runner that creates our objects. Getting dependencies in a test is so awkward because we do not know if all objects are registered, this requires waiting a certain grace period. However, there is a little bundle out there that can use the DS annotations to emulate some (really some, it is only supporting configuration and the `@Reference` annotation) of DS's behavior. This is the aQute.dstest bundle. It can be used in standard JUnit (without an OSGi framework) but it can also work with the framework when given a Bundle Context.

Add this bundle to your build path:

```
-runbundles: \
    cnf.run.base, \
    aQute.dstest;version=1.1.0, \
    osgi.enroute.blog.memory.provider;version=latest
```

Let's see how this works:

```
BlogManager                                  blog;
@Override
public void setUp() throws Exception {
    DummyDS ds = new DummyDS();
    ds.setContext(
      FrameworkUtil.getBundle(BlogTest.class).getBundleContext());
    ds.add(this);
    ds.wire();
}
```

The `setUp` method is called by JUnit before a test is run. In this case we add our own object to the DummyDS object as well as the Bundle Context. The `wire` method will make sure the `@Reference` annotations are filled in. `DummyDS` will look at the requested type and inject an appropriate object. It will look

at any of the added objects, classes, or any strings. the `add(String)` method will attempt to load a class out of any of the resolved bundles if a Bundle Context was registered (after no object or class is found locally).

Any parameters on the `@Reference` annotation, like the `target` filter, are ignored (I told you, it is limited).

```
BlogManager blog;

@Reference
void setBlogManager(BlogManager bm) {
  this.blog = bm;
}
public void testSimple() {
  System.out.println(blog);
}
```

Select the testSimple method name, call up the context menu, and select Debug As/Bnd OSGi Test Launcher (JUnit). The console will show:

```
osgi.enroute.blog.memory.provider.BlogManagerImpl@6cf56dcc
Tests run  : 1
Passed     : 1
Errors     : 0
Failures   : 0
```

We will no longer use the `testSimple` method so best remove it.

Testing the API

Since we have a Blog Manager service now, we can start testing it. Since the Blog Manager provides CRUD operations, we create a `testCrud` method. Let's first make it create a post.

```
public void testCrud() throws Exception {
  long now = System.currentTimeMillis();
  long id;
  try {
    BlogPost p = new BlogPost();
    p.title = "Title";
    p.content = "text";
    id = blog.createPost(p);
  } catch (Exception e) {
    e.printStackTrace();
    fail();
    return;
  }
}
```

Then we test if we could actually create this post:

```
 try {
    BlogPost post = blog.getPost(id);
    assertNotNull(post);
    assertThat(post.content, is("text"));
    assertThat(post.title, is("Title"));
    assertTrue(post.created >= now);
    assertTrue(post.created <= System.currentTimeMillis());
    assertTrue(post.lastModified >= now);
    assertTrue(post.lastModified <= System.currentTimeMillis());
    long created = post.created;
```

Now we update the post:

```
    BlogPost p = new BlogPost();
    p.id = id;
    p.content = "other";
    p.title = "OtherTitle";
    blog.updatePost(p);
```

And verify that we really updated it:

```
      post = blog.getPost(id);
      assertNotNull(post);

      assertThat(post.content, is("other"));
      assertThat(post.title, is("OtherTitle"));
      assertThat(post.created, is(created));
      assertTrue(post.lastModified <= System.currentTimeMillis());
      assertTrue(post.lastModified >= created);
    } catch (Exception e) {
      e.printStackTrace();
      fail();
    }
```

Wonderful! A green bar!

We have not tested the `queryBlogs` method. We can add this test at the end of the the testCrud method. We could make a separate test but then we would have to add records to see anything in the list. So we can use the state of the previous tests. Though in general you have to be careful to not rely on state in unit tests, these test are actually quite cohesive so we can do it this way.

```
      fail();
    }
    try {
      int n = 0;
      Iterator<BlogPost> i = blog.queryBlogs(null).iterator();
      while (i.hasNext()) {
        i.next();
        n++;
      }
      assertEquals(1, n);
    } catch (Exception e) {e.printStackTrace(); fail();}
  }
```

Obviously, this test won't work if we had a database. The test has the assumption that the database is empty when the test methods start.

Making it a Test Bundle

We can run the test inside Eclipse and that is great for debugging. However, we also need to run test in a continuous integration system, or just from the command line. This is very well supported by bnd, and since bnd runs in maven, ant, gradle, make, sbt, or a myriad of other build tools, it is well supported in any build environment.

However, when we run this outside the user interface we do not know which class we should run as tests; in the user interface the users point out the classes to test by selecting a project, a package, a class, or a method. Therefore, bnd has designated a header: `Test-Cases`. This header is used by bnd in a non-user interface situation to run all listed test cases.

Obviously, it is painful to maintain this header by hand, though you could. Fortunately there is a macro in bnd that will enumerate classes depending on super class, annotations, name, etc. In cnf/build.bnd you find a macro that uses this ${classes} macro to enumerate any JUnit TestCase classes. This macro looks like:

```
test-cases: ${classes;CONCRETE;EXTENDS;junit.framework.TestCase}
```

What is says is that any class in the bundle that is concrete, and somewhere extends `TestCase` is considered a test class. This works well for JUnit 3 that requires all test cases to extend the TestCase class. In Junit 4, for some unknown reason, annotations have made this more, well, *flexible*. You can also use the `classes` macro to test for annotations. However, we use the good old JUnit 3 way here although we use some of the JUnit 4 features.[7]

We can therefore automatically fill the `Test-Cases` header in the `bnd.bnd` file. go to the Source tab of the bnd.bnd file and add:

```
Test-Cases:             ${test-cases}
```

We can now run the tests from the command line. If you have the Google Console installed (if not, you should), select the test project, then call up the context menu and select: `Open Terminal Here`. This will create a console in the directory of the project. You can now run:

[7] Support for JUnit 4 is not yet well tested. Use at your own risk.

```
$ ant test
[..
[.......... lots of (too much actually) logging
[...
Testing /Ws/osgi.enroute.blog/osgi.enroute.blog.test/bnd.bnd
[bndtest] ____________________________
[bndtest] Welcome to Apache Felix Gogo
[bndtest]
[bndtest] Tests run  : 1
[bndtest] Passed     : 1
[bndtest] Errors     : 0
[bndtest] Failures   : 0
[bndtest] All tests passed
```

If bnd is installed (you can install bnd with `sudo jpm install bnd@*`) on the command line, you can also run:

```
$ bnd test
____________________________
Welcome to Apache Felix Gogo

g!
Tests run  : 1
Passed     : 1
Errors     : 0
Failures   : 0
No Errors
```

The output can be garbled because we also start the shell, its prompt can be at a random place.

References

- OSGi Framework Util
  http://www.osgi.org/javadoc/r4v43/core/org/osgi/framework/FrameworkUtil.html
- aQute.dstest
  http://jpm4j.org/#!/p/osgi/aQute.dstest
- bnd ${classes} macro
  http://www.aqute.biz/Bnd/Macros#classesx

11. JPA Implementation (Very Experimental)

Goal

Use JPA to store the Blog Posts in a relational database.

Prerequisites

First ensure that you have checked out the following branch in the workspace.

```
$ git checkout 09-api-tester
```

You should compare the the project against your own solution of the previous step. you can also continue with your previous step. If you continued from the previous phase then the framework should still be running. Verify this; if it is not running then start it. We will update the framework incrementally.

Background

Disclaimer: The JPA support is experimental. Don't assume the future will look like anything described in here. This support bundles from the `https://github.com/osgi/osgi.bundles` repository are currently alpha quality.

JPA has taken the Java world by storm, although it has left some wreckage in its wake. It is a specification that is as sweet as a siren's song because it makes using relational databases look simple. Unfortunately, Object Relational mapping is more astrology than astronomy due to the infamous impedance mismatch between the object model (objects point to their members) and the relational model (members point to their containers via foreign keys). Maybe this could have been manageable if JPA had been thoroughly specified and accompanied with a solid TCK. Ah well.

Enough soap box. JPA is an important technology and the support of it in OSGi is abominable. Though the OSGi specified a JDBC Data Source Factory and a JPA specification, support from the market has been minimal. There is one known implementation of the Data Source Factory (h2), and no known implementations of JPA that work as advertised; both Hibernate and EclipseLink need significant plumbing in OSGi and exhibit wildly different behavior for the same persistence unit. The Apache Aries project tries to provide this in their Aries *container*. The Eclipse Gemini project provides an OSGi environment for Eclipse Link. Apache OpenJPA is interesting but only supports JPA 2.0, and does not seem eager to upgrade (surprisingly, JPA 1.0 to 2.0 seems minor from an API perspective, while 2.0 to 2.1 looks quite large. So much for semantic versioning).

In the OSGi enRoute project an attempt is made to create a pluggable JPA environment. In this model a consumer of the JPA specification contains one or more persistence unit XML files in its bundle. These persistence units are picked up, combined with a Data Source configured by Configuration Admin, and then turned into an Entity Manager service. This Entity Manager service uses the Transaction (JTA) to make it thread safe for the caller, i.e. all bundles share the same Entity Manager but this EM proxies to a delegate per thread. This is also called a managed environment in the JPA specification; it is based on the SPI interface in JPA.

The enRoute JPA project intention is to go away over time because the available implementations support this functionality out of the box. However, in this step we are going to take a look at it. Note that this is so far experimental.

This step will create a provider of the `BlogManager` API based on JPA. It will use the test suite to test it, and then create a bndrun configuration to let the Blog Application run against a database.

JPA Provider

Create a new project, `osgi.enroute.blog.jpa.provider`. In this project we will use the test bundle to test our implementation. This means that we need to add the test bundle to our run bundles, as well as all its dependencies. We will have to add transaction support to the test case since the JPA Managed model requires it. Let us go through the `bnd.bnd` file.

First, the main headers. Version, description. We implement our code in the `osgi.enroute.blog.jpa.provider` package, and just as the memory provider, we export and provide the API package.

```
Bundle-Version:         1.0.0.${tstamp}
Bundle-Description:             Implement the BlogManager API on JPA.
Private-Package: osgi.enroute.blog.jpa.provider
Export-Package: osgi.enroute.blog.api.*;provide:=true
```

Now the dependencies to build. We have collected some of the common dependencies in the `cnf.api` library (see `cnf/libs` and the Libs repository). This includes transactions, connector, jpa, and OSGi. We also need bnd's annotations, the Blog Manager API, and JUnit.

```
-buildpath: \
    cnf.api;version='[1,2)', \
    biz.aQute.bnd.annotation;version=2.2,\
    osgi.enroute.blog.api;version=latest, \
    org.apache.servicemix.bundles.junit;version='[4.11.0,5.0.0)'
```

This should be sufficient to write the implementation.

Domain Class

The Blog Manager API puts us a bit in a bind. Generally JPA courses assume you own the domain objects and allow you to freely sprinkle annotations on the fields and malign the class in other ways. However, with a service based model, where the service is defined by a *contract*, the domain objects are not yours. Since you cannot sprinkle annotations all over the place, we will need a placeholder. Additional constraints[8] in JPA implementations require us to create a new Blog Post class that uses getters setters and private fields to enable the JPA providers to weave this class to do its magic. Obviously, this means that our implementation using JPA must closely track the `BlogPost` domain object from the API.[9]

[8] It is not clear (to me) in the spec that JPA requires getters and setters but this is definitely required in Eclipse Link and Hibernate. Would love to stand corrected.

[9] If someone comes up with a better solution ... let me know, but remember we cannot touch the domain object class.

Create a `osgi.enroute.blog.jpa.provider.BlogPostImpl` class. This class must mimic the `BlogPost` domain class exactly, however, here we can apply the JPA annotations!

```
@Entity
public class BlogPostImpl {
  @Id
  private long          id;
  private long          created;
  private long          lastModified;
  private String title;
  private String content;

  BlogPost getBlogPost() {
    BlogPost post = new BlogPost();
    post.id = getId();
    post.content = getContent();
    post.lastModified = getLastModified();
    post.created = getCreated();
    post.title = getTitle();
    return post;
  }

  // setters/getters go here
}
```

As stated before, we must add the crufty getters and setters. These are not shown in the following code, use Eclipse's `Source/Generate Getters and Setters` menu to create your getters and setters.

Once we have this handler class for Blog Post domain objects, the Blog Manager is straightforward. First thing we want to add is the name of the table. This is in general good practice and improves the portability.

```
@Entity
@Table(name = "POST")
public class BlogPostImpl {
```

In our memory provider we could use an Atomic Long to assign the `id`. With a database, that might be used concurrently, there is no simple solution. For some reason that escapes me, SQL has no standard way to generate sequence numbers although virtually all known database implement such a feature in wildly or very subtle different ways. One of the advantages of JPA should be that it supports sequences regardless of JPA provider and database. After several mishaps that had did not work on either EclipseLink or Hibernate I found the following way we could generate a sequence number for the id. Surprisingly, the numbers are different between Eclipse and Hibernate. Eclipse starts at 1001 (one higher than specified and Hibernate starts a 100). Ah well, specs are highly overrated. Anyway, other setups differed even more.[10]

```
@Table(name = "POST")
@SequenceGenerator(name = "blogPostIds",
    initialValue = 1000, allocationSize = 100)
public class BlogPostImpl {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
       generator = "blogPostIds")
    private long id;
```

During testing it became clear that one of the JPA providers supported very long `content` while the other stopped at 255 characters. It was therefore necessary to specify the length with an annotation.

```
private String   title;

@Column(length = 100000)
private String   content;
```

persistence.xml

JPA requires an XML file that defines one ore more *persistence units*. The location(s) of this XML must be pointed out by the `Meta-Persistence` header in the bundle's manifest. Generally this file is used to define the persistence provider, the database driver, the database name etc. However, in the setup for managed OSGi JPA we do not defined any of those details in the persistence unit since this would make it hard to change during deployment. In the current model, the database is specified using Configuration Admin. The persistence.xml is therefore quite small, especially since we only have a single class we persist.

[10] I am no JPA expert, so please provide feedback how this can be so unportable?

```
<persistence
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/
ns/persistence/persistence_2_0.xsd"
  version="2.0">
  <persistence-unit name="tutorial" transaction-type="JTA">
    <class>osgi.enroute.blog.jpa.provider.BlogPostImpl</class>
   </persistence-unit>
</persistence>
```

We place this class in the project directory and must then include it in the `-includeresource` instruction in the `bnd.bnd` file. Since we also need to include it in the `Meta-Persistence` manifest header, the `bnd.bnd` file should contain the following clauses:

```
Meta-Persistence: persistence.xml
-includeresource: \
    {${Meta-Persistence}}
```

You can verify it gets included in the root of the bundle by double clicking `generated/osgi.enroute.blog.jpa.provider`.

The Manager Implementation

The implementation of the Blog Manager is the class `osgi.enroute.blog.jpa.provider.BlogManagerImpl`. It uses, as could be expected, the DS component support.

```
@Component
public class BlogManagerImpl implements BlogManager {

  private EntityManager em;

  @Reference
  void setEM(EntityManager em) {
    this.em = em;
  }
}
```

Getting a Blog Post is usually the most straight forward:

```
private EntityManager   em;

@Override
public BlogPost getPost(long id) throws Exception {
  BlogPostImpl post = getPostImpl(id);
  if (post == null)
    return null;
  return post.getBlogPost();
}
private BlogPostImpl getPostImpl(long id) {
  try {
    BlogPostImpl post = (BlogPostImpl) em
      .createQuery("SELECT p FROM BlogPostImpl p WHERE p.id=:id")
      .setParameter("id", id).getSingleResult();
    return post;
  } catch (NoResultException e) { return null;}
}
```

Deleting a post is almost as simple:

```
@Override
public void deletePost(long postId) throws Exception {
  BlogPostImpl post = getPostImpl(postId);
  if (post == null)
    throw new FileNotFoundException("No such post");
  em.remove(post);
}
```

Create and update are very similar, they share a common method.

```
@Override
public long createPost(BlogPost content) throws Exception {
  BlogPostImpl post = new BlogPostImpl();
  post.setCreated(System.currentTimeMillis());
  update(content, post);
  return post.getId();
}
@Override
public void updatePost(BlogPost content) throws Exception {
  BlogPostImpl old = getPostImpl(content.id);
  if (old == null)
    throw new FileNotFoundException("No such post");
  update(content, old);
}
```

And the shared method:

```
private void update(BlogPost from, BlogPostImpl to) {
  to.setContent(from.content);
  to.setTitle(from.title);
  to.setLastModified(System.currentTimeMillis());
  em.persist(to);
  em.flush();
}
```

The hardest one is the search method. It should be possible to do the search in the database but my JQL skills failed. We therefore use a dummy method, improvements welcome.

```
@Override
public Iterable<BlogPost> queryBlogs(String query) throws Exception {
  List<BlogPostImpl> posts = em.createQuery(
    "SELECT p FROM BlogPostImpl p").getResultList();
  List<BlogPost> result = new ArrayList<>();
  Pattern filter = null;
  if (query != null && !posts.isEmpty()) {
    query = query.trim();
    query = query.replace("*", ".*").replace("?", ".?");
    query = "(" + query.replaceAll("\\s+", ")|(") + ")";
    filter = Pattern.compile(query, Pattern.CASE_INSENSITIVE);
  }
  for (BlogPostImpl p : posts) {
    if (filter == null || filter.matcher(p.getContent()).find()
        || filter.matcher(p.getTitle()).find())
      result.add(p.getBlogPost());
  }
  return result;
}
```

readme.md

Don't forget to include a readm.md for others ... In the bnd.bnd file:

```
-includeresource: \
    {${Meta-Persistence}}, \
    {readme.md}
```

And you can write the `readme.md` yourself now ...

Adapting the Test Case

The JPA provider requires transactions, this is implicit in the contract of OSGi Managed JPA. The OSGi Managed JPA provides a servlet filter that begins a transaction for each servlet request. However, in the test case we do not pass the servlet engine. We therefore need to create transaction boundaries. This requires adapting the test cases to begin and end transactions properly. So the following code changes are about the `osgi.enroute.blog.test.BlogTest` class.

First we must add a build path dependency on the transaction manager, JTA. The JTA API is in the cnf.api library in the Libs repository.

```
-buildpath: \
    osgi.core;version='[4.3.1,6)',\
    org.apache.servicemix.bundles.junit;version='[4.11.0,5.0.0)', \
    aQute.dstest;version=1.1.0, \
    cnf.api;version='[1,2)', \
    osgi.enroute.blog.api;version=latest
```

We can get the Transaction Manager with the `@Reference` annotation. We need to declare a field to hold the transaction manager.

```
BlogManager                   blog;
private TransactionManager    tm;
```

And add the dependency:

```
@Reference
void setBlogManager(BlogManager bm) {
  this.blog = bm;
}
@Reference
void setTransactionManager(TransactionManager tm) {
  this.tm = tm;
}
```

We then need to add the tm.begin(), tm.commit(), and tm.rollback() methods around all our database interactions. Shown is the first part of the test case. Adding the other units of work is left as an exercise. The try/catch blocks should give a hint where they should be applied.

```
tm.begin();
long id;
try {
  BlogPost p = new BlogPost();
  p.title = "Title";
  p.content = "text";
  id = blog.createPost(p);
  tm.commit();
} catch (Exception e) {
  tm.rollback();
  e.printStackTrace();
  fail();
  return;
}
```

Since we now need a transaction manager, we also need to make some fixes that this **`osgi.enroute.blog.test`** project can run the tests locally. These changes are also necessary for the JPA provider, they are mainly involved because we now require a transaction manager. They will be explained in more detail there. These are the changes necessary for the `osgi.enroute.blog.test` `bnd.bnd` file.

```
-runpath: \
  org.apache.servicemix.bundles.junit;version=4.11, \
  org.apache.geronimo.specs.geronimo-jta_1.1_spec;version=1.1.1

-runbundles: \
  aQute.dstest;version=1.1.0, \
  osgi.enroute.blog.memory.provider;version=latest, \
  cnf.run.base, \
  org.apache.geronimo.specs.geronimo-j2ee-connector_1.6_spec,\
  cnf.org.ow2.jotm
```

You can now run the `osgi.enroute.blog.test` project as `Run As/Bnd OSGi Test Launcher (JUnit)` again.

Running (that is Testing)

The JPA provider project does not have to run stand alone since it is not an application. However, during development it is nice to run the code to verify the implementation is correct. The best way to do this is to prepare that this JPA provider project can be run as an OSGi Test project. We can do this by including the `osgi.enroute.blog.test` project and making sure that our run environment is correct for running this bundle. So we need to create a run environment that satisfies our needs as well as the needs of the test environment. We therefore need to include the `aQute.dstest` bundle used in the test bundle. We further add some debug bundles like the shell and X-Ray. While testing they can be life savers and it does not cost anything to add them. In this configuration we add Hibernate and H2. You can easily switch between Hibernate and EclipseLink, switch the `cnf.org.hibernate` entry for `cnf.eclipse.persistence`. Both entries are libraries, look in `cnf/libs` and the Libs repository.

```
-runbundles: \
    cnf.run.base, \
    cnf.run.web, \
    cnf.run.web.debug, \
    aQute.dstest;version=1.1.0, \
\
    osgi.enroute.blog.test;version=latest, \
\
    cnf.run.persistence, \
    cnf.org.hibernate, \
    org.h2
```

The Java JRE and Java EE have a bizarre mismatch in the contents of the `javax.transaction` package. The JRE is missing a number of types. Since these classes come from the class path, they cannot easily be overridden by bundles. People have tried to do this. Since the JRE packages are normally not well versioned, they try to use 1.1 for the Java EE packages (which is the official specification version). Well, this generally result in a big mess and *uses constraints violations*.If you do not know what that means, you like to keep it that way. Fortunately, bnd can place JARs on the run time class path. It will analyze these JARs and uses any `Export-Package` clauses to add these packages as if they were exported by the System bundle. This not only gives us a correct version for these packages, it also fills in the missing classes. Unfortunately, it cannot override class from the boot class path (this is impossible in Java), but at least we can make it workable. Since we only use this project in test mode (we do not run it normally) we can add these JARs therefore to the `-testpath`. The `-testpath` has the same functionality as the `-runpath` but is only used for testing. The `-runpath` is used for both situations. We also need to add the JUnit JAR since it is mandatory that the JUnit runner (which does not run as a bundle) uses the same classes as the bundles.

```
-testpath: \
    org.apache.servicemix.bundles.junit;version=4.11, \
    org.apache.geronimo.specs.geronimo-jta_1.1_spec;version=1.1.1
```

There is one more thing we need to correct in the VM before it can run the application. Hibernate requires the `javax.xml.stream` packages to be versioned. These packages are provided by the VM but are not versioned. The following sets up version for these packages:

```
-runsystempackages: \
    javax.xml.stream;version=1.1, \
    javax.xml.stream.events;version=1.1, \
    javax.xml.stream.util;version=1.1
```

The OSGi Managed JPA setup requires a Data Source Factory to get an XA Data Source for the database and it requires a configuration for the JPA Manager to allow it to create an Entity Manager service.

We could of course add a `configuration/configuration.json` file. However, the `aQute.configurer` bundle also supports a property for configuration: `bnd.configuration`. The value of this property is a JSON string. This is a very convenient way to specify small configurations. That said, don't make them too large since it is hard in a **`bnd.bnd`** file to get them right because you have to add the backslashes for each new line. For this setup, we allow the JPA manager to manage all found persistence units. For the datasource we create an in-memory database in H2.

```
-runproperties: bnd.configuration = '[ \
    {\
        "service.factoryPid" : "osgi.jpa.managed.aux.JPAManager",\
        "service.pid" : "Wildcard",\
        "name" : "*"\
    }, {\
        "service.factoryPid" : "osgi.jdbc.managed.aux.XADataSourceFactory",\
        "service.pid" : "H2DataSource",\
        "url" : "jdbc:h2:mem:db",\
        "dSF.target" : "(osgi.jdbc.driver.class=org.h2.Driver)"\
    }\
]'
```

It is now finally time to run the test! Select the project, call up the context menu and select `Debug As/Bnd OSGi Test Launcher (JUnit)`. This runs the test suite against our new implementation.

Command Line

If we try to run the test from the command line then ant will not run the test because this project is not a test project, ant ignores no-test projects. The reason is that bnd does not detect a `Test-Cases` header. Fortunately we can force it with the `-f/--force` flag:

```
$ bnd test --force
 _____________________________
Welcome to Apache Felix Gogo

g! 25-10-13 20:49:06 (I) Jotm.<init> : RMI initialization
25-10-13 20:49:06 (I) Jotm.<init> : JOTM started with a local transaction factory

.... lots of logging

[EL Finest]: 2013-10-25 20:49:07.969--ServerSession(1183027056)--
Connection(1260875737)--Thread(Thread[main,5,main])--Connection released to connection
pool [read].
Tests run  : 1
Passed     : 1
Errors     : 0
Failures   : 0
No Errors
$
```

ant has a similar problem but in this case we can not set the `-f` option. There is therefore a special flag that can be set to make the test run anyway. This propery is `project.osgi.junit`. Setting this property to true in the `bnd.bnd` file will make ant run the test anyway.

```
project.osgi.junit = true
```

Adapting the Application

Ok, we have the test running. However, our application is still using the cheap memory provider. Obviously we could change the `bnd.bnd` file in the application project to use the JPA provider instead of the memory provider but then we would always have to test against the JPA provider, which is by its nature much more complex and heavy than the Memory provider.

We therefore create a `bndrun` file. A `bndrun` file is a full specification for an application, it defines the same runtime environment instructions as in the `bnd.bnd` file. Let's create an `enblog.bndrun` file in the application project. The New menu has a special entry for this `New/Run Descriptor`. Make it an empty run descriptor so we can discuss each step.

Understanding the setup of the JPA provider run environment should make it clear what the different instructions do, and why they are necessary. They are very similar. The interesting difference is the addition of the `osgi.jta.filter.aux` bundle. This bundle will register a filter that begins a transaction for each web request and commits it when the request has been executed (or rolls it back when there was an exception). This is the reason we do not have to adapt our application to know transactions.

```
-runbundles: \
    cnf.run.base, \
    cnf.run.web, \
    cnf.run.web.debug, \
    osgi.enroute.blog.appl;version=latest, \
    aQute.twitter.bootstrap;version=2.3.2,\
    aQute.google.angular.stable;version=1.0.8,\
    osgi.enroute.blog.jpa.provider;version=latest, \
    cnf.run.persistence, \
    osgi.jta.filter.aux;version=1.0.0, \
    cnf.org.eclipse.persistence, \
    org.h2

-runpath: \
    org.apache.geronimo.specs.geronimo-jta_1.1_spec;version=1.1.1

-runsystempackages: \
    javax.xml.stream;version=1.1, \
    javax.xml.stream.events;version=1.1, \
    javax.xml.stream.util;version=1.1
```

You can now select the `enblog.bndrun` file, call up the context menu, and select `Run As/Bnd OSGi Run Launcher`. This will launch a framework and run the application. Don't hesitate to try it out with the Hibernate bundles (`cnf.org.hibernate`) instead of the Eclipse persistence.

We added the configuration necessary to create the Entity Manager and the corresponding XA Data Source as a property in the `bnd.bnd` file. Since we now run from the enblog.bndrun file, we will not see that property. We could add the property in here but we decide to add it to the `configuration/configuration.json` file. In a situation where these services are not installed the configuration records do not harm.

```
        "alias": "/rest"
    }, {
        "service.factoryPid" : "osgi.jpa.managed.aux.JPAManager",
        "service.pid" : "Wildcard",
        "name" : "*"
    }, {
        "service.factoryPid" : "osgi.jdbc.managed.aux.XADataSourceFactory",
        "service.pid" : "H2DataSource",
        "url" : "jdbc:h2:mem:db",
        "dSF.target" : "(osgi.jdbc.driver.class=org.h2.Driver)"
    }, {
            "service.pid" : "osgi.jta.filter.aux.JTACoordinator"
    }
]
```

If you double click on the `enblog.bndrun` file and select the Run tab then you can see a `Run OSGi` and `Debug OSGi` buttons at the right top. Make sure you have no framework running already, then click on one of these buttons. Go to http://localhost:8080/#/ and refresh to test.

Packaging

We now have a nice application with full JPA storage of our endeavors of writing poetic or prosaic blogs. Would it not be cool if we could run this as an application from the command line?

bnd has a `package` command that can create an *executable JAR*. This is a self contained JAR file with a manifest that contains a `Main-Class` header. This header points to the class with a `main` method. Such a JAR can be executed in java with the `-jar` option. bnd will add everything needed to run the application, including the framework, the launcher and all the needed bundles. Complete.

Creating an executable JAR requires that we document it. Just like a bundle, this JAR can escape to the world and then it is always opportune to document and identify it. We therefore add some documentation to do in the `enblog.bndrun` file.

```
Bundle-Version:  1.0.0.${tstamp}
Bundle-Description:    A simplistic blog application to show of en Route. \
 The application is a web service with a GUI to list, add, delete, and
 update blogs. Just go to port http://localhost:8080 and enjoy.

Include-Resource: {readme.md}
```

As you might have noticed, you were invited to write another `readme.md` file to explain what this JAR actually does. We can now run the bnd package command on this file and run it.

```
$ bnd package enblog.bndrun
$ ls *.jar
enblog.jar
$ java -jar enblog.jar
_____________________________
Welcome to Apache Felix Gogo

g!
```

Since this JAR contains all its dependencies, it is very easy to deploy it on for example Amazon or Cloud foundry. However, `java -jar` ... would it not be nice if you could run the application from the command line, just under the name `enblog`? This is the purpose of the jpm command (Just Another Package Manager for Java). If the JAR file has a JPM-Command header then jpm can install it on the command line.

```
Bundle-Description:    A simplistic blog application to show of en Route. \
 The application is a web service with a GUI to list, add, delete, and
 update blogs. Just go to port http://localhost:8080 and enjoy. This \
 command can be  installed with jpm under the name ${JPM-Command}.

JPM-Command:                enblog
```

We can now package it and install it on the command line.

```
$ bnd package enblog.bndrun
$ sudo jpm install enblog.jar
$ enblog
Welcome to Apache Felix Gogo

g!
```

The command can be placed on jpm4j, which will make it available world wide. Anybody can then install it with:

```
$ jpm candidates enblog
```

References

- h2
  http://www.h2database.com/html/main.html
- Hibernate
  http://www.hibernate.org/
- Eclipse Link
  http://www.eclipse.org/eclipselink/
- Apache Aries
  http://aries.apache.org/
- Eclipse Gemini
  http://www.eclipse.org/gemini/
- Apache OpenJPA
  http://openjpa.apache.org/
- OSGi enRoute Bundles
  https://github.com/osgi/bundles
- Uses Constraints
  http://njbartlett.name/2011/02/09/uses-constraints.html
- Configuration Admin
  http://www.osgi.org/javadoc/r4v42/org/osgi/service/cm/ConfigurationAdmin.html