Instructions for the enRoute Blog Tutorial

Peter Kriens

Table Of Content

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 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 that all files are UTF-8.
4. From the market place, install bndtools. bndtools is a plugin that provides extensive support for OSGi development.
5. Ensure git is installed: http://git-scm.com/book/en/Getting-Started-Installing-Git
6. Install jpm: http://jpm4j.org/#!/md/install
7. Install bnd: `sudo jpm install bnd@*`
8. 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`

2. Setup: Running a Framework

Goal

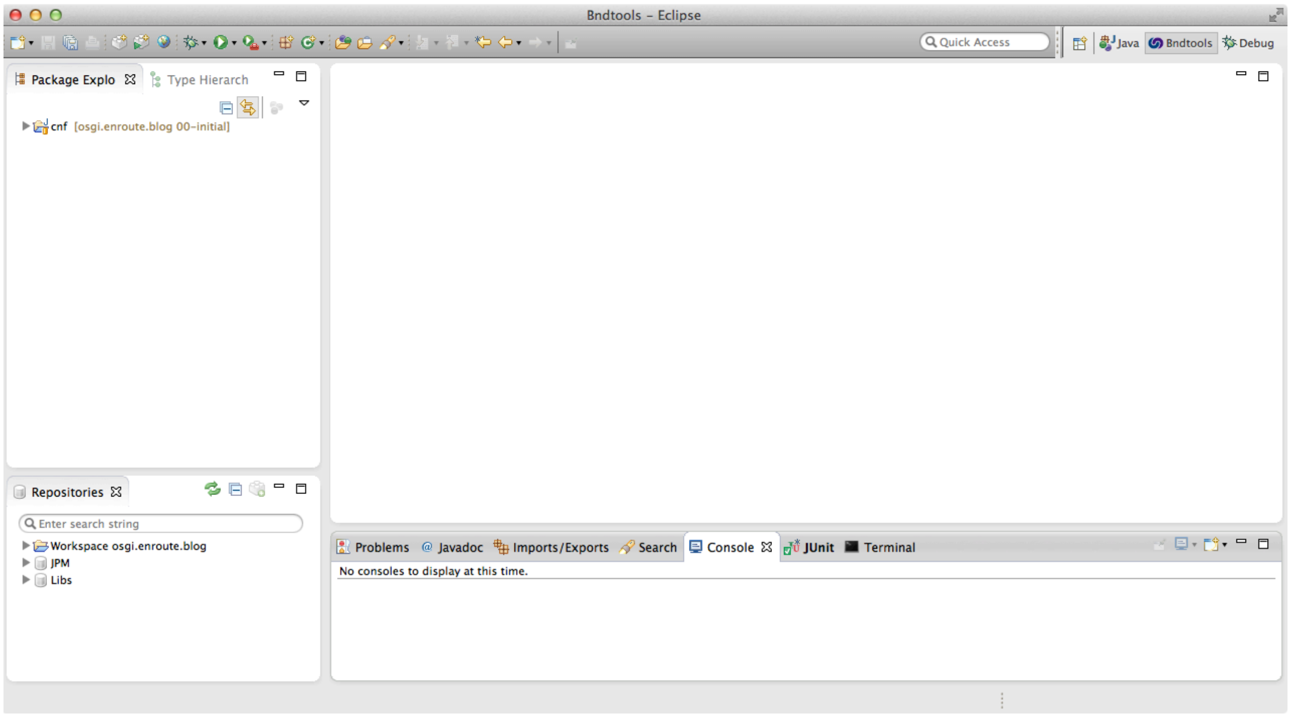
To get a framework running, inspecting what tools are available for us to develop and thus debug applications, and provide configuration data from a project.

Prerequisites

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

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

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

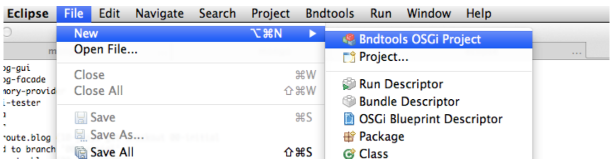


No consoles to display at this time.

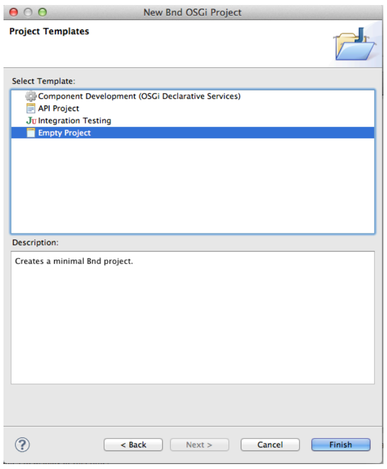
The only project is `cnf`, this directory defines the setup of the workspace and provides shared information like build files, etc.

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 template so choose an empty project.



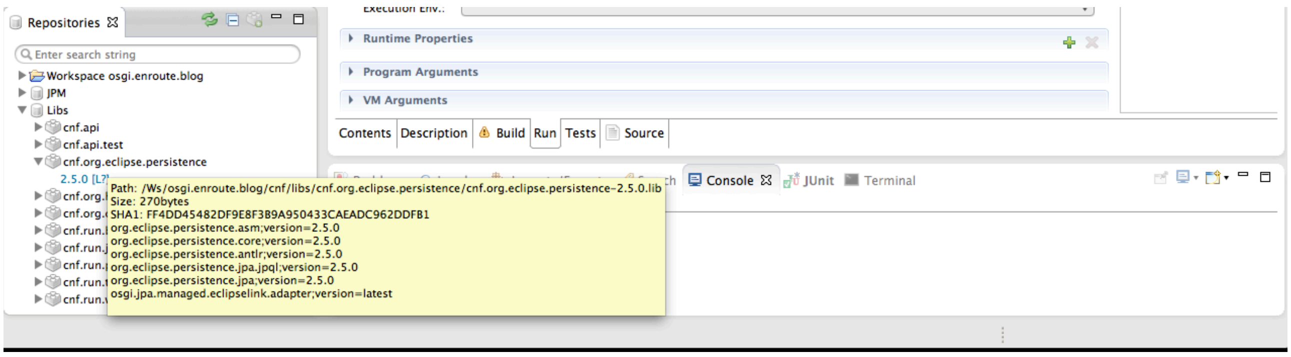
Setting up

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}
```

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.



```
Path: /Ws/osgi.enroute.blog/cnf/libs/cnf.org.eclipse.persistence/cnf.org.eclipse.persistence-2.5.0.lib
Size: 270bytes
SHA1: FF4DD45482DF9E8F3B9A950433CAEADC962DDFB1
org.eclipse.persistence.asm;version=2.5.0
org.eclipse.persistence.core;version=2.5.0
org.eclipse.persistence.antlr;version=2.5.0
org.eclipse.persistence.jpa.jpql;version=2.5.0
org.eclipse.persistence.jpa;version=2.5.0
osgi.jpa.managed.eclipselink.adapter;version=latest
```

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).

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

The GUI is a shell over the bnd.bnd text file. Click on 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 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, bnd will raise a warning if we try to run it now since there is no content. So create a readme.md file with a text editor to describe the bundle and include it in the bundle:

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

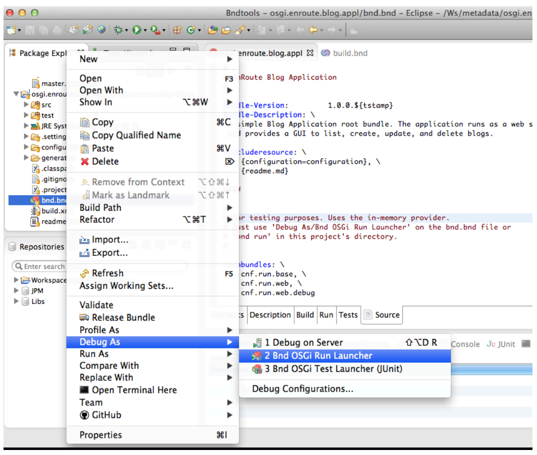
The curly braces around readme.md tell bnd to preprocess the text file, this means you can use any macro. In general, you like to run text files used in the build through the pre-processor.

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

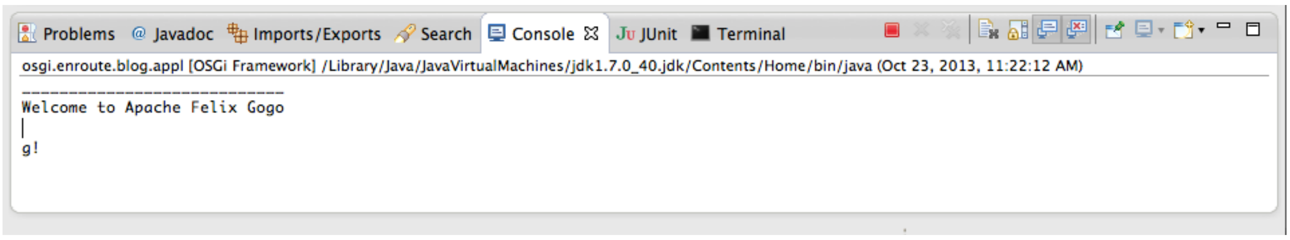
This together is sufficient to run this, admittedly rather useless, bundle. However, it will show us the setup of the framework. Select the bnd.bnd file and call up the context menu. Then select `Debug As/Bnd OSGi Run Launcher`.



```
Route Blog Application

lle-Version:        1.0.0.${tstamp}
lle-Description: \
imple Blog Application root bundle. The application runs as a web s
l provides a GUI to list, create, update, and delete blogs.

luderesource: \
{configuration=configuration}, \
{readme.md}

r testing purposes. Uses the in-memory provider.
st use 'Debug As/Bnd OSGi Run Launcher' on the bnd.bnd file or
nd run' in this project's directory.

bundles: \
cnf.run.base, \
cnf.run.web, \
cnf.run.web.debug
```

Gogo Shell

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



```
osgi.enroute.blog.appl [OSGi Framework] /Library/Java/JavaVirtualMachines/jdk1.7.0_40.jdk/Contents/Home/bin/java (Oct 23, 2013, 11:22:12 AM)
____________________________
Welcome to Apache Felix Gogo

g!
```

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:

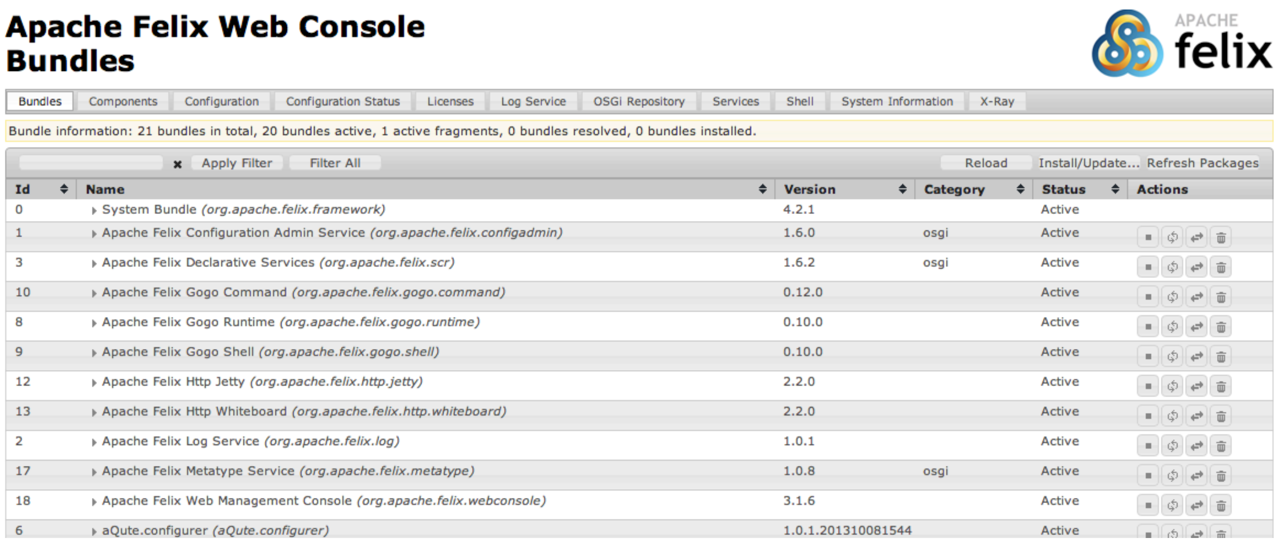
- `help` — A list with all commands
- `help <scope:command>` — Help for a specific command, e.g. `help scr:list`
- `bundles` — List all bundles (this is actually directly calling `BundleContext.getBundles`).
- `start <n>` — Start bundle n (also `(bundle <n>) start`)
- `stop <n>` — Stop bundle n (also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

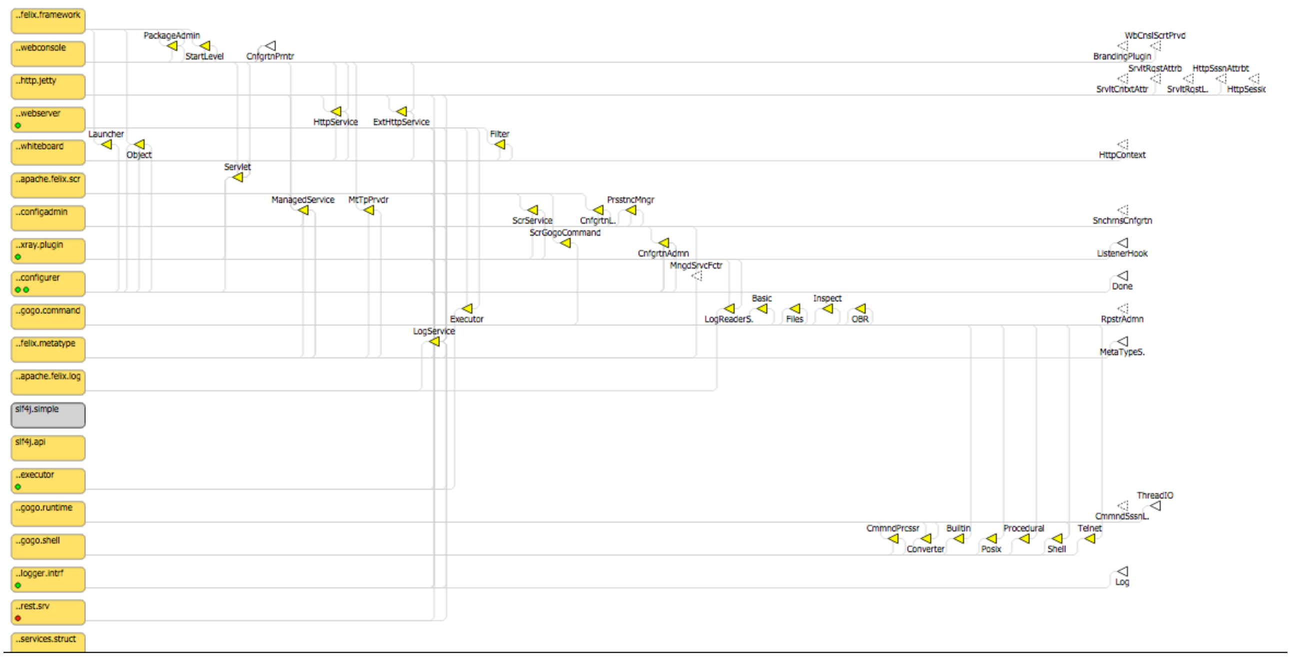


Apache Felix Web Console
Bundles

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

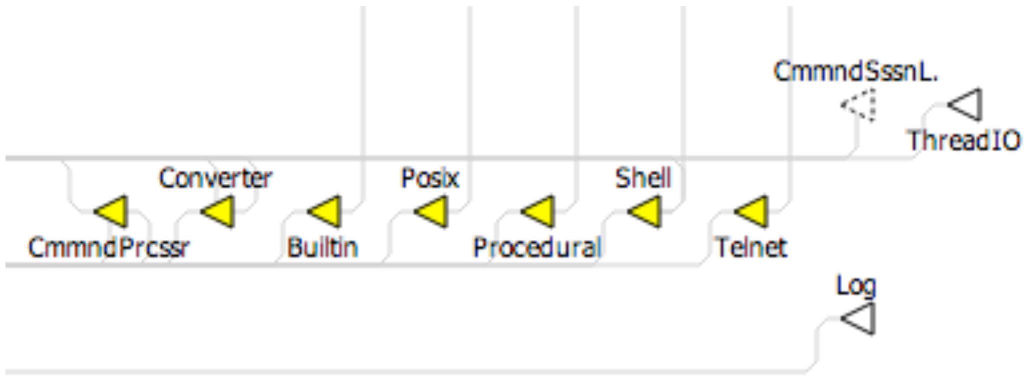
Id	Name	Version	Category	Status	Actions
0	System Bundle *(org.apache.felix.framework)*	4.2.1		Active	
1	Apache Felix Configuration Admin Service *(org.apache.felix.configadmin)*	1.6.0	osgi	Active	
3	Apache Felix Declarative Services *(org.apache.felix.scr)*	1.6.2	osgi	Active	
10	Apache Felix Gogo Command *(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.

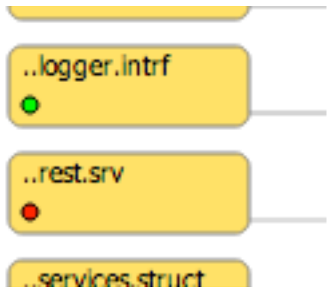


Bundles display their state through color coding. Orange is `ACTIVE`, red is `STARTING`, white is `INSTALLED`, and `RESOLVED` is gray. You can click on the bundle's icon to go to the Bundle's tab in the Web Console at the given bundle. Hovering over the bundle icon shows you the exact name.

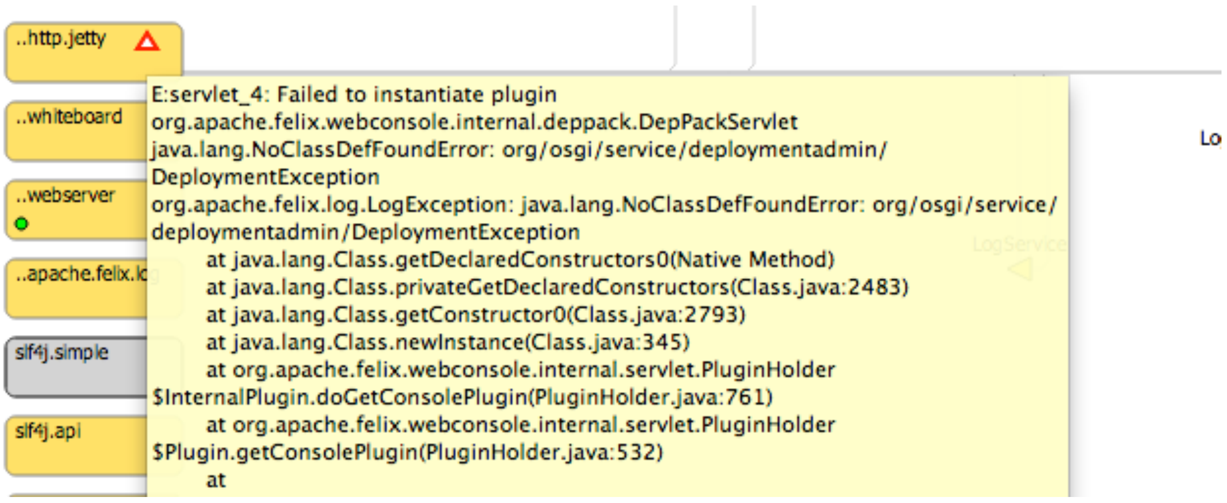
Services are yellow if they are in use. A white service with a black outline registered but not used, when the outline is dashed there is a bundle looking for it. The white services are displayed at the right. If something is not working, look at these services because they often indicate that bundles/components 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.



```
E:servlet_4: Failed to instantiate plugin
org.apache.felix.webconsole.internal.deppack.DepPackServlet
java.lang.NoClassDefFoundError: org/osgi/service/deploymentadmin/
DeploymentException
org.apache.felix.log.LogException: java.lang.NoClassDefFoundError: org/osgi/service/
deploymentadmin/DeploymentException
        at java.lang.Class.getDeclaredConstructors0(Native Method)
        at java.lang.Class.privateGetDeclaredConstructors(Class.java:2483)
        at java.lang.Class.getConstructor0(Class.java:2793)
        at java.lang.Class.newInstance(Class.java:345)
        at org.apache.felix.webconsole.internal.servlet.PluginHolder
$InternalPlugin.doGetConsolePlugin(PluginHolder.java:761)
        at org.apache.felix.webconsole.internal.servlet.PluginHolder
$Plugin.getConsolePlugin(PluginHolder.java:532)
        at
```

X-Ray is full dynamic, it will continuously poll the framework and update its status.

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
    }
]
```

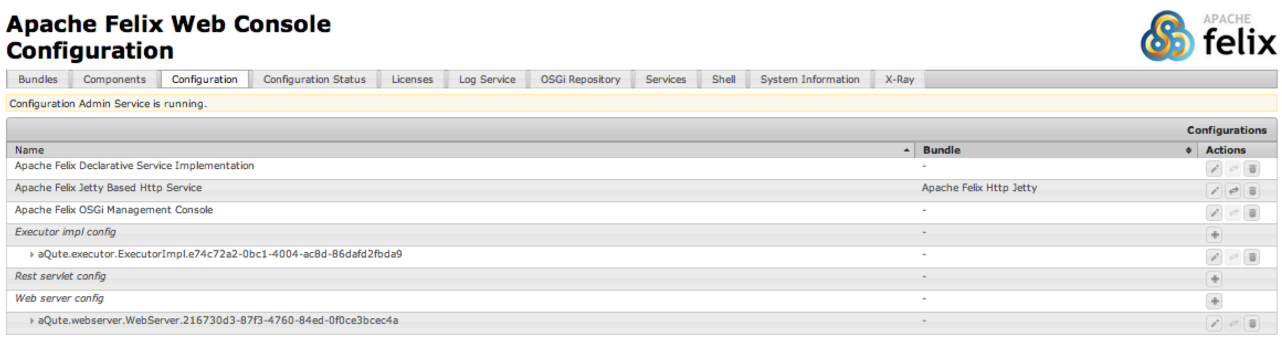
This provides configurations for the Apache Felix http server, the Webserver (provides extender support for bundles delivering content) and an Executor shared by all bundles.

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 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):

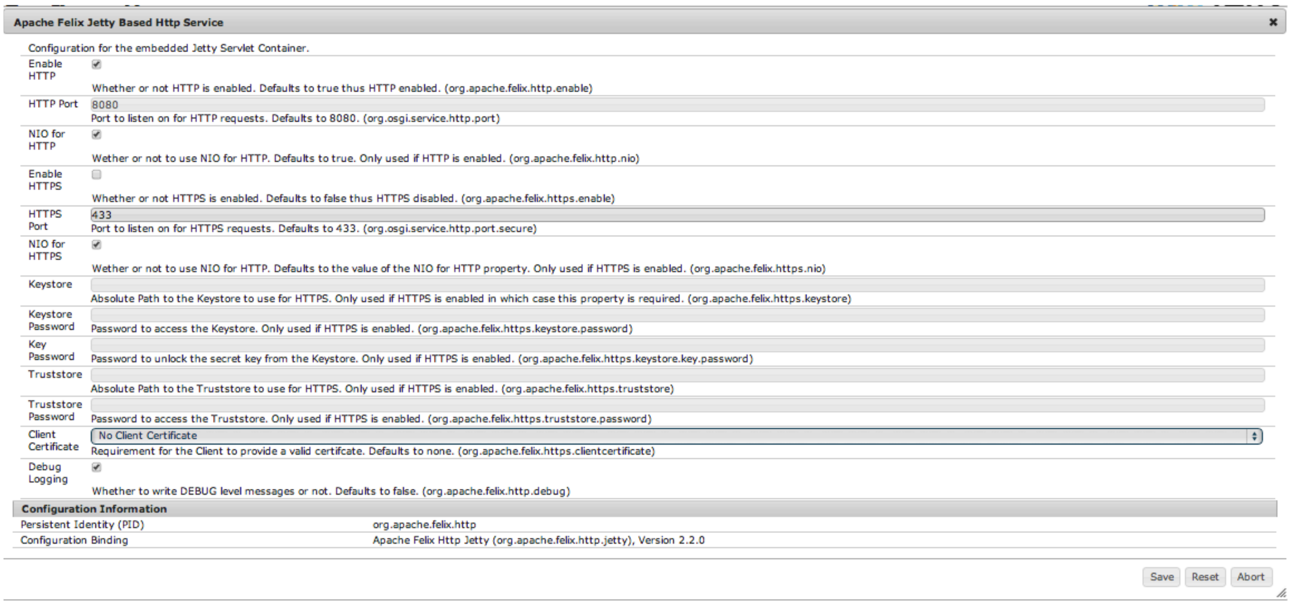


Configuration Admin Service is running.

Configurations

Name	Bundle	Actions
Apache Felix Declarative Service Implementation	-	
Apache Felix Jetty Based Http Service	Apache Felix Http Jetty	
Apache Felix OSGi Management Console	-	
Executor impl config	-	
aQute.executor.ExecutorImpl.e74c72a2-0bc1-4004-ac8d-86dafd2fbda9	-	
Rest servlet config	-	
Web server config	-	
aQute.webserver.WebServer.216730d3-87f3-4760-84ed-0f0ce3bcec4a	-	

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.



Apache Felix Jetty Based Http Service

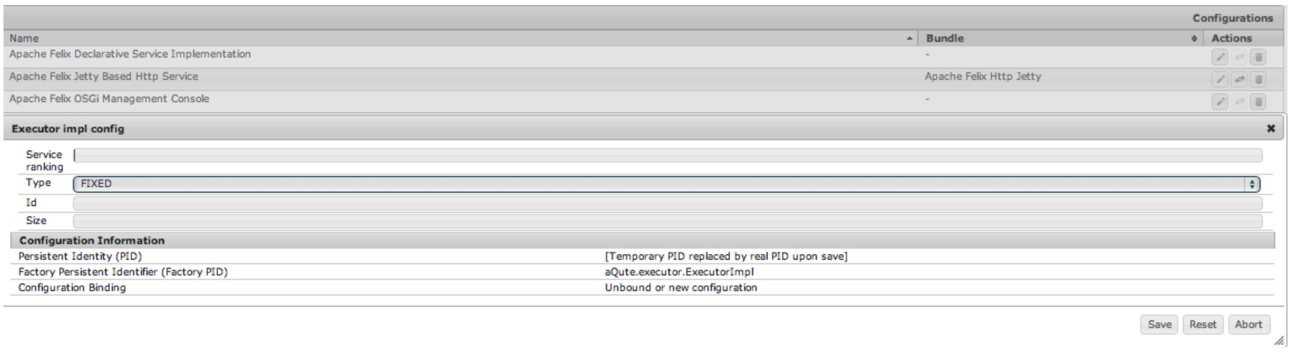
Configuration for the embedded Jetty Servlet Container.

Enable HTTP	✓ Whether or not HTTP is enabled. Defaults to true thus HTTP enabled. (org.apache.felix.http.enable)
HTTP Port	8080 Port to listen on for HTTP requests. Defaults to 8080. (org.osgi.service.http.port)
NIO for HTTP	✓ Wether or not to use NIO for HTTP. Defaults to true. Only used if HTTP is enabled. (org.apache.felix.http.nio)
Enable HTTPS	☐ Whether or not HTTPS is enabled. Defaults to false thus HTTPS disabled. (org.apache.felix.https.enable)
HTTPS Port	433 Port to listen on for HTTPS requests. Defaults to 433. (org.osgi.service.http.port.secure)
NIO for HTTPS	✓ Wether or not to use NIO for HTTP. Defaults to the value of the NIO for HTTP property. Only used if HTTPS is enabled. (org.apache.felix.https.nio)
Keystore	Absolute Path to the Keystore to use for HTTPS. Only used if HTTPS is enabled in which case this property is required. (org.apache.felix.https.keystore)
Keystore Password	Password to access the Keystore. Only used if HTTPS is enabled. (org.apache.felix.https.keystore.password)
Key Password	Password to unlock the secret key from the Keystore. Only used if HTTPS is enabled. (org.apache.felix.https.keystore.key.password)
Truststore	Absolute Path to the Truststore to use for HTTPS. Only used if HTTPS is enabled. (org.apache.felix.https.truststore)
Truststore Password	Password to access the Truststore. Only used if HTTPS is enabled. (org.apache.felix.https.truststore.password)
Client Certificate	No Client Certificate Requirement for the Client to provide a valid certifcate. Defaults to none. (org.apache.felix.https.clientcertificate)
Debug Logging	✓ Whether to write DEBUG level messages or not. Defaults to false. (org.apache.felix.http.debug)

Configuration Information

Persistent Identity (PID)	org.apache.felix.http
Configuration Binding	Apache Felix Http Jetty (org.apache.felix.http.jetty), Version 2.2.0

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:



Configurations

Name	Bundle	Actions
Apache Felix Declarative Service Implementation	-	
Apache Felix Jetty Based Http Service	Apache Felix Http Jetty	
Apache Felix OSGi Management Console	-	

Executor impl config

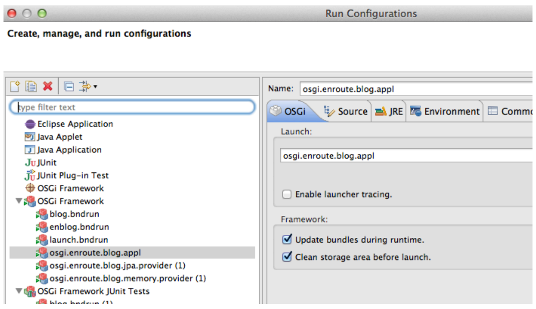
Service ranking	
Type	FIXED
Id	
Size	

Configuration Information

Persistent Identity (PID)	[Temporary PID replaced by real PID upon save]
Factory Persistent Identifier (Factory PID)	aQute.executor.ExecutorImpl
Configuration Binding	Unbound or new configuration

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`.



Run Configurations

Create, manage, and run configurations

Name: osgi.enroute.blog.appl

Launch: osgi.enroute.blog.appl

☐ Enable launcher tracing.

Framework:

☑ Update bundles during runtime.

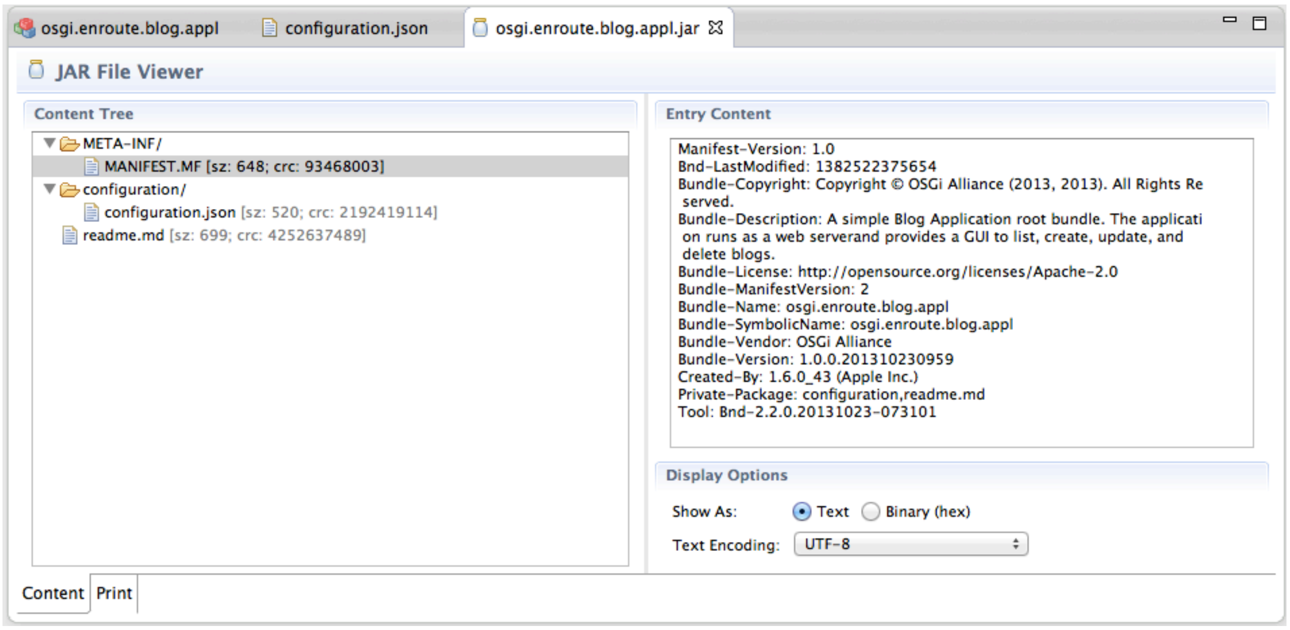
☑ Clean storage area before launch.

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. This will open the JAR viewer.



```
Manifest-Version: 1.0
Bnd-LastModified: 1382522375654
Bundle-Copyright: Copyright © OSGi Alliance (2013, 2013). All Rights Re
 served.
Bundle-Description: A simple Blog Application root bundle. The applicati
 on runs as a web serverand provides a GUI to list, create, update, and
 delete blogs.
Bundle-License: http://opensource.org/licenses/Apache-2.0
Bundle-ManifestVersion: 2
Bundle-Name: osgi.enroute.blog.appl
Bundle-SymbolicName: osgi.enroute.blog.appl
Bundle-Vendor: OSGi Alliance
Bundle-Version: 1.0.0.201310230959
Created-By: 1.6.0_43 (Apple Inc.)
Private-Package: configuration,readme.md
Tool: Bnd-2.2.0.20131023-073101
```

Show As: [✓ Text] Binary (hex)
Text Encoding: UTF-8

As you can see, it includes a Manifest, the configuration, and the readme.md file.

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
$ cd osgi.enroute.blog.appl
```

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 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 /index.html.

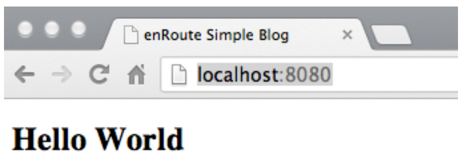
Adding a web page is therefore as simple as creating a file in the static directory. In this phase the only thing we will do is add an `static/index.html` file with the following content:

```
<!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:



Hello World

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

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
$ cd osgi.enroute.blog.appl
```

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

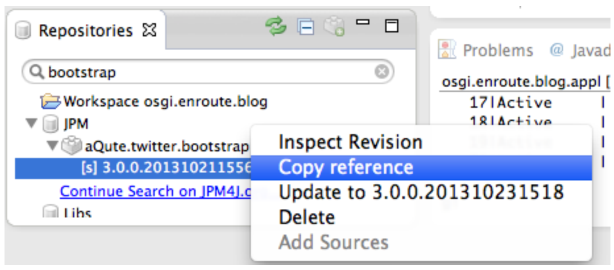
Cascading Style Sheets (CSS) are by far the most (unnecessary) complex and mind blowing deficient web technology. In any project, the web styling can create a huge drain on productivity and, unless there are experts, results in mediocre pages. That was, until Twitter brought us Bootstrap, a CSS framework. 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 css is used, it will look like thousands of other websites).

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 and search for `bootstrap` in the search input field. This will show you the bundle in the repository. 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 the text pane.

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 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.) 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>
```

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.

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.

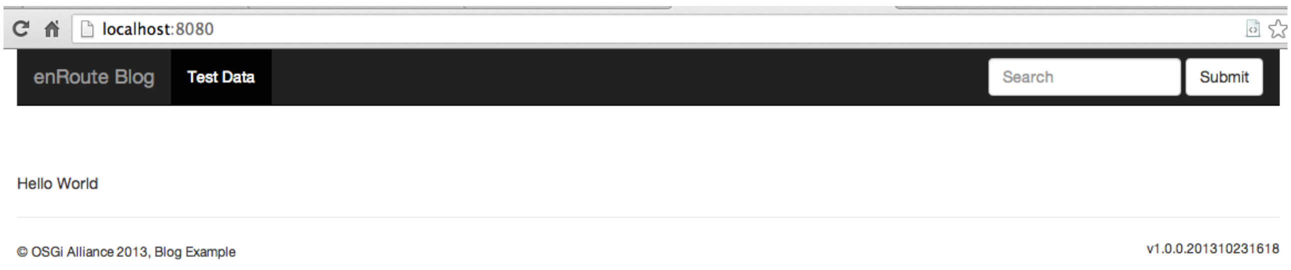
```
<div style="margin-top: 60px;"></div>
Hello World
<hr>
```

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 and enjoy the result:



Hello World

© OSGi Alliance 2013, Blog Example v1.0.0.201310231618

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

To familiarize oneself with the basic operation of Angular JS.

Prerequisites

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

```
$ git checkout 03-bootstrap
$ cd osgi.enroute.blog.appl
```

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 user the different concerns in such an application. Smalltalk came up with the Model, View, Controller (MVC) division in the seventies of the last century and this model is still seen as the primary way architect user interface applications. Unfortunately, most MVC implementations fall far short since it almost always requires some code to update the view when the model changes. Not so in Angular, 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 (silly) Blog Application in Javascript. It shows a form with an editor for a blog post and its title, and a list of 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 in the JPM repository. You should also add this to the `-runbundles` in the `bnd.bnd` file. Which will then 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
```

This makes Angular available under the `/google/angular/angular.js` path. We therefore need to 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.

Minimal

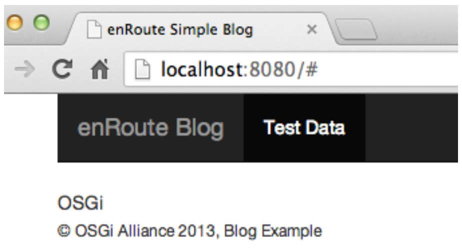
To get started, we need to tell Angular it can touch the DOM of the page, this requires an `ng-app` attribute at a tag, any content inside that tag is then free game for Angular. In our case, we add the `ng-app` attribute at the `html` tag since the whole page is an Angular application.

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

Now change the Test Data button in the navigation bar so that it sets the `hello` variable in the model by adding an `ng-click` attribute.

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

Though the syntax looks Javascript code it is actually restricted; Angular does have its own compiler. Any variables referred in this context are stored in the *scope*, in this case we only have a root scope. We can now refer to this variable. The simplest way to see the variable is to use the double curly braces , like`{{hello}}`. You can insert this in place of the Hello World text. 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'.



OSGi
© OSGi Alliance 2013, Blog Example

Module

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

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 be nested.

A controller is a 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 work, we create a 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)
```

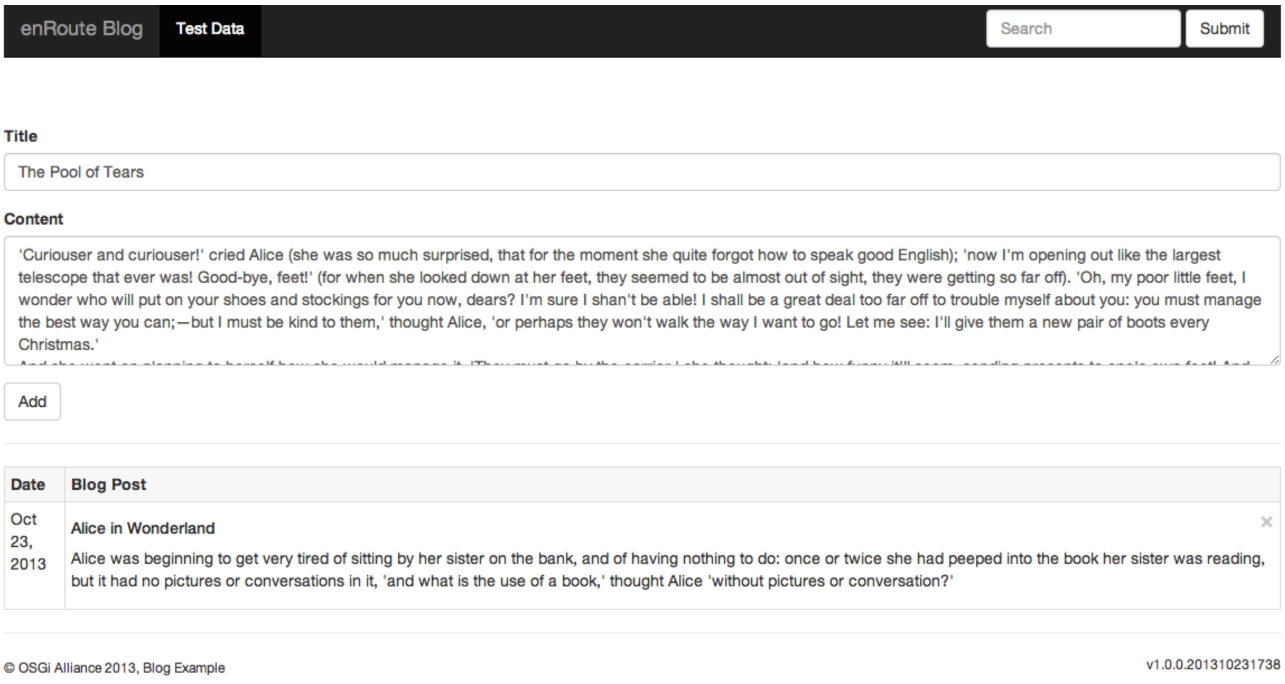
We can associate the top level controller at the `body` tag of the `index.html` file. We can use the Javascript name, `enBlog` since we made it a global variable. (In Javascript all `window`’s properties are global variables.)

```
</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 show 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.



Title

The Pool of Tears

Content

'Curiouser and curiouser!' cried Alice (she was so much surprised, that for the moment she quite forgot how to speak good English); 'now I'm opening out like the largest telescope that ever was! Good-bye, feet!' (for when she looked down at her feet, they seemed to be almost out of sight, they were getting so far off). 'Oh, my poor little feet, I wonder who will put on your shoes and stockings for you now, dears? I'm sure I shan't be able! I shall be a great deal too far off to trouble myself about you: you must manage the best way you can;—but I must be kind to them,' thought Alice, 'or perhaps they won't walk the way I want to go! Let me see: I'll give them a new pair of boots every Christmas.'

Add

Date	Blog Post
Oct 23, 2013	**Alice in Wonderland** ✕ Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do: once or twice she had peeped into the book her sister was reading, but it had no pictures or conversations in it, 'and what is the use of a book,' thought Alice 'without pictures or conversation?'

© OSGi Alliance 2013, Blog Example v1.0.0.201310231738

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.

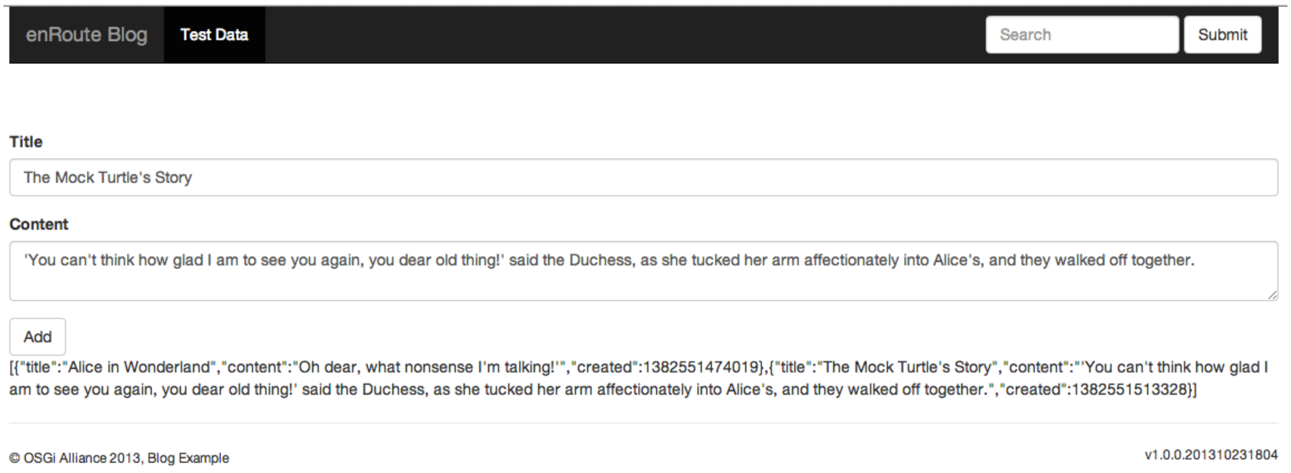
```
<button ng-click=save() type=”submit” class=”btn btn-default”>Add</button>
```

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, and refresh the browser! Type some text and press the add button.



Title

The Mock Turtle's Story

Content

'You can't think how glad I am to see you again, you dear old thing!' said the Duchess, as she tucked her arm affectionately into Alice's, and they walked off together.

[{"title":"Alice in Wonderland","content":"Oh dear, what nonsense I'm talking!'","created":1382551474019},{"title":"The Mock Turtle's Story","content":"'You can't think how glad I am to see you again, you dear old thing!' said the Duchess, as she tucked her arm affectionately into Alice's, and they walked off together.","created":1382551513328}]

© OSGi Alliance 2013, Blog Example

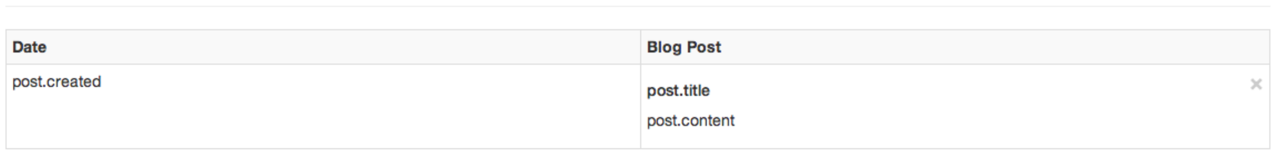
v1.0.0.201310231804

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.

```
<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:



Date	Blog Post
post.created	× post.title post.content

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 a scope expression giving the collection to enumerate. For example: `post in posts`.

The to be repeated tag is the `tr` tag, holding 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 carefull 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. 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 ...
```

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.

```
<style>[ng-click] { cursor: pointer; }</style>
```

Deleting an Entry

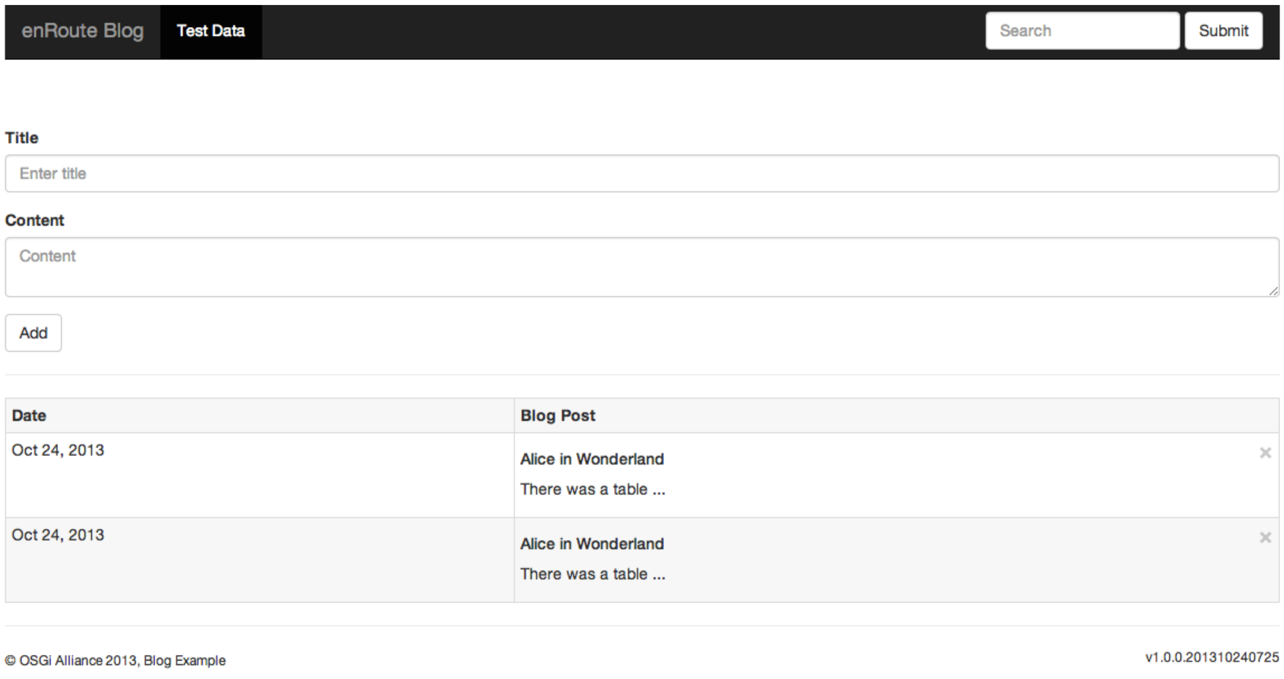
In the table we are showing a × (`&times; \u2A09`) symbol on the right top to delete an entry. However, for now this entry is not deleted. So lets add this behavior. First add the function call to the button. Call the function `remove`, and pass it the `post` as argument.

```
        <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. You first get the index of the element and then splice the array on that position. This 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 ...



Date	Blog Post
Oct 24, 2013	Alice in Wonderland There was a table ...
Oct 24, 2013	Alice in Wonderland There was a table ...

© OSGi Alliance 2013, Blog Example

v1.0.0.201310240725

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

6. REST Calls

Goal

Communicate to the back-end server through REST calls.

Prerequisites

First ensure that you have checked out the ‘Angular’ branch in the workspace.

```
$ git checkout 04-angular
$ cd osgi.enroute.blog.appl
```

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 is convenient, it is not a compelling reason to use OSGi. We 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. Methods on the created objects then provide the CRUD operations. There are also a number of factory methods to query the server, get instances, save an instance, and remove an instance.

A 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 values. That is, return values must in general be an object, primitive values like numbers and strings do not work.

On the back-end side 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 Resource Manager services. A Resource Manager is a service that offers a number of public methods as 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 normally 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.
- 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();
}
```

The the method `BlogPost putBlogpost(SaveOptions options,long id)` maps to:

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

All transfers are done in JSON using public fields where the data type is mapped to the applicable JSON type.

Add Dependency to the $resource Module

The `$resource` service comes from the `ngResource` module, it is not built into Angular. We need to make two changes to make this new module available to our own `Blog` module. The first change is in the index.html, 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>
```

In Angular, it is necessary to explicitly declare module dependencies, we therefore need to fill in the previously mysterious list in `blog.js`'s module declaration.

```
(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).

This has only added the dependencies, this has not changed any behavior net.

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 variable:

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

We can remove the `EXAMPLE` variable since we will get the contents from the server.

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 to `index.html` so we can see if we get a result. 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.

```
</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.

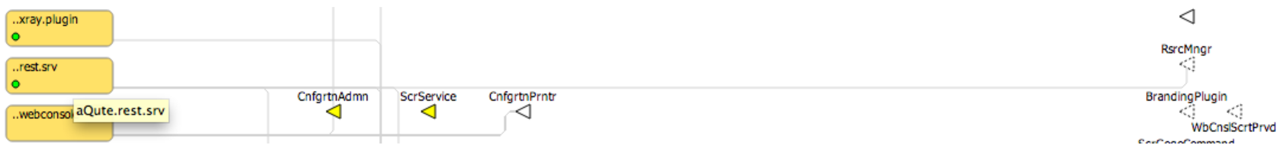
Starting the REST server

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 (which is picked up by the Apache Felix Whiteboard support) and it listens for Resource Manager 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 String    content;
    public String    title;

    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;
  }
}
```

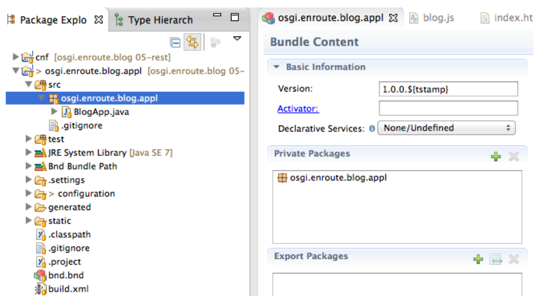
This class imports the `ResourceManager` class and the `Options` interface, however, they are not on the *build path*. The build path is maintained in the `bnd.bnd` file, in the `-buildpath` instruction. These classes are in the aQute.rest.srv bundle, you can find this bundle in the JPM repository, just search for it. The Component annotation comes from bnd and can be found in the biz.aQute.bnd.annotation 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 only 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 test data 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.



```
cnf [osgi.enroute.blog 05-rest]
> osgi.enroute.blog.appl [osgi.enroute.blog 05-
  src
    osgi.enroute.blog.appl
      BlogApp.java
    .gitignore
  test
  JRE System Library [Java SE 7]
  Bnd Bundle Path
  .settings
  > configuration
  generated
  static
  .classpath
  .gitignore
  .project
  bnd.bnd
  build.xml
```

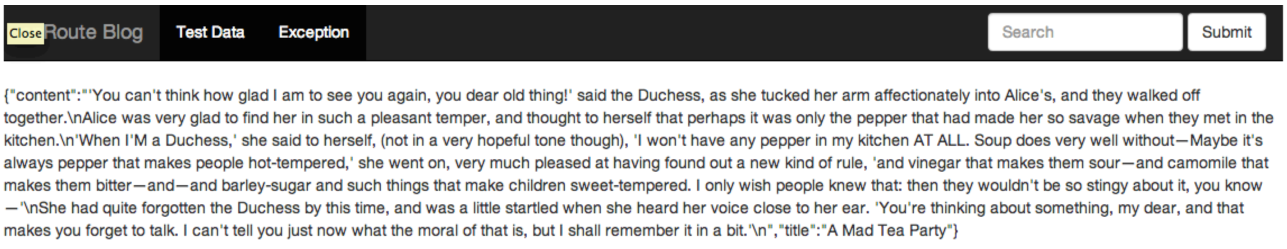
Version: 1.0.0.${tstamp}
Activator:
Declarative Services: None/Undefined

Private Packages: osgi.enroute.blog.appl

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 think how glad I am to see you again, you dear old thing!' said the Duchess, as she tucked her arm affectionately into Alice's, and they walked off together.\nAlice was very glad to find her in such a pleasant temper, and thought to herself that perhaps it was only the pepper that had made her so savage when they met in the kitchen.\n'When I'M a Duchess,' she said to herself, (not in a very hopeful tone though), 'I won't have any pepper in my kitchen AT ALL. Soup does very well without—Maybe it's always pepper that makes people hot-tempered,' she went on, very much pleased at having found out a new kind of rule, 'and vinegar that makes them sour—and camomile that makes them bitter—and—and barley-sugar and such things that make children sweet-tempered. I only wish people knew that: then they wouldn't be so stingy about it, you know —'\nShe had quite forgotten the Duchess by this time, and was a little startled when she heard her voice close to her ear. 'You're thinking about something, my dear, and that makes you forget to talk. I can't tell you just now what the moral of that is, but I shall remember it in a bit.'\n","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.

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. 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. Thy can then be passed to the REST calls. In preparation, we make these functions available to 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 class names and its value is a `boolean` (well for Javascript everything is a `boolean`) that indicates if the given class name should be present or not in the `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>
```

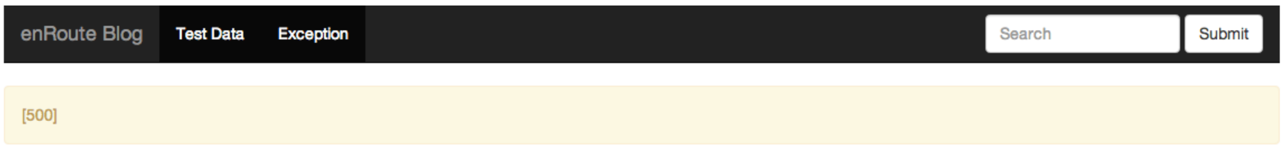
Last, but not least we need to change to the `BlogApp` class. First we add a new `EXCEPTION` field in the `Command` enum.

```
  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.



[500]

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 ‘Angular’ branch in the workspace.

```
$ git checkout 05-rest
$ cd osgi.enroute.blog.appl
```

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` page would soon become unwieldy if we tried to cram all HTML in there. However, 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 functions as well as bookmarking. Solid applications use the URI of the page to make the application navigable.

8. Blog Backend

9. Service Based Design

10. Testing

11. JPA Implementation

12. Packaging