Restful Objects 1.0 Users Guide

RESTful Web Services for Naked Objects 4.0.x

Version 0.1

Preface

Restful Objects is a sister project to Naked Objects and provides an implementation of a web-based viewer that exposes a RESTful web service for a Naked Objects domain model.

This manual describes how to use the *Restful Objects* in both prototype mode and how to deploy it into production as a webapp. It also describes how to use *Restful Objects*' applib to write client-side applications.

Restful Objects is hosted on SourceForge, and is licensed under Apache Software License v2. Naked Objects is also hosted on SourceForge, and is also licensed under Apache Software License v2.

Chapter 1

Introduction

Restful Objects is a Naked Objects viewer implementation to expose a domain model using REST. So let's briefly explain what REST is, and why you might want to use it.

1.1. Introducing REST

Web services were introduced as a means for different computer systems to interact by network even though they may be implemented in different technologies and with different implied domain models. SOAP is probably the best well-known protocol for doing this, though there have been others. These days there is a whole bunch of specifications over and above those for SOAP; together these are typically called WS-*.

Whatever; what characterizes the WS-* implementations is that they expose only a single endpoint to be invoked; in effect just another way of doing a remote procedure call. Put another way, WS-* -style web services provide an for a verb - "do this for me". In fact, this RPC endpoint usually accepts many different message types, and uses some sort of dispatcher to route the message so it can be processed correctly.

REST (standing for *REpresentational State Transfer*) in contrast is a style of designing web services that is modelled on the human web sites. Rather than expose a single endpoint, it exposes multiple endpoints. And these endpoints don't represent verbs, they represent things (or nouns). But REST goes further than this, because it also restricts what we can do with those resources to the standard HTTP verbs: GET (read), PUT (update), DELETE (er, delete) and POST (invoke, or change in some way). The first three of these are idempotent (can be invoked multiple times with no side effects). This is an important characteristic for building scalable systems, and is not one that the RPC approach towards web services supports at all well.

What happens if we perform an HTTP GET on a resource? Well, we get a representation of that resource. REST doesn't mandate what that representation is, but typical choices are JSON, XHTML or a custom XML dialect. However, what REST does emphasise is that these resources should be linked together, again analogous to the way that the human web works with hyperlinks.

So REST is much more closely associated with the web - and HTTP in particular - than WS-* -style web services ever were. But - so the thinking goes - the vast majority of web services are deployed over the web, so why disregard the web's semantics?

For much more on the design principles and philosophy behind REST, you might want to read Richardson & Ruby's RESTful Web Services.

1.2. Introducing Restful Objects

As already stated, *Restful Objects* is a Naked Objects viewer implementation to expose a domain model using REST. That is, the resources are the actual domain objects, or - in some cases - a class member of one of those objects. The following table shows the resources exposed:

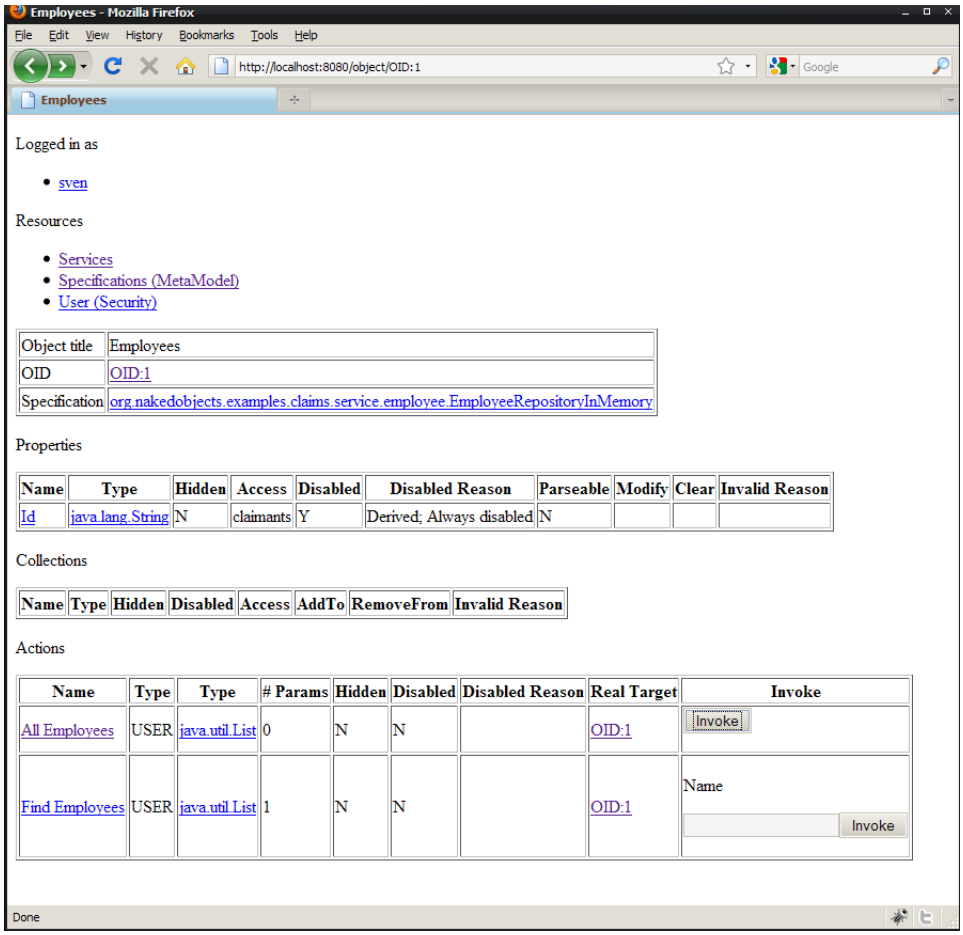
Table 1.1. Verbs by Resources

Verb \ Resource	Object	Property	Collection	Action
GET	current state of all properties	n/a	current contents	n/a
PUT	create	set	add to	n/a
DELETE	remove	clear	remove from	n/a
POST	n/a	n/a	n/a	invoke

Restul Objects exposes a URL for each of the columns. So, for example:

- a `Customer` instance id=12345 might be exposed as `http://localhost:8080/object/CUS|12345`, where "CUS|12345" is the (string representation of) the internal Naked Objects object identifier (or Oid) that uniquely identifies the domain object. Performing a GET on this would list all of the properties of the `Customer`.
- the `orders` collection for this same `Customer` would be exposed as `http://localhost:8080/object/CUS|12345/collection/orders`. Performing a GET on this would list (links to) the contents of the collection.

So much for interacting with the resources, but what of their representation? Well, (following the suggestion in Richardson & Ruby's book) *Restful Objects* renders the domain objects using XHTML. This gives us a natural way to express links between resources: we just use hyperlinks using the `<a href="..."/>` tag. It also means that we can inspect the domain objects from an web browser. For example, here's Firefox displaying the resource representing an `EmployeeRepository` (part of of the example claims application that is in the Naked Objects Maven download):



In fact, *Restful Objects* also serves up a smidgeon of Javascript as well, to allow the web browser to perform PUT and DELETE as well as the usual GET and POST. This won't be needed by your own custom written web apps, of course. (It also won't be needed once XHTML 5 - with its support for these additional HTTP verbs - becomes mainstream).

Restful Objects itself is implemented using JBoss RestEasy.

1.3. Limitations of REST

Because REST is newer than the original WS-*-style services, it currently lacks some supporting features. For example, some of the supporting WS-* specifications deal with such things as security credentials and transaction propagation. At the time of writing REST is yet to gain these sorts of additional support. That said, *Restful Objects* provides some basic support for security, however.

Chapter 2

Using Restful Objects in Prototypes

Exposing a RESTful domain model with *Restful Objects* is really pretty straightforward; we just need to boot Naked Objects to use the *Restful Objects*' viewer rather than the DnD viewer or HTML viewer that comes out-of-the-box. That means adding references to the POM, and the running with the appropriate command line flags.

If you ran the Naked Objects archetype then you'll have a Maven parent module with a number of child modules:

```
xxx/pom.xml
    xxx-dom/pom.xml          # domain object model
    xxx-fixture/pom.xml      # fixtures for seeding object store
    xxx-service/pom.xml      # in-memory object store implementations of repositories
    xxx-commandline/pom.xml  # for deploying as a commandline, also for prototyping
    xxx-webapp/pom.xml       # for deploying as a webapp
```

The instructions here assume this directory structure.

2.1. Parent Module

In the parent module, first add in a `<properties>` section to specify the version of *Restful Objects*. This will transitively bring in any dependencies:

```
<properties>
  <restfulobjects.version>1.0.0</restfulobjects.version>            <!-- or whatever -->
</properties>
```

Then, add in references to the *Restful Objects*' `applib` and `viewer` modules to the `<dependencyManagement>`:

```
<dependencyManagement>
  <dependencies>
    ...
    <dependency>
      <groupId>org.starobjects.restful</groupId>
      <artifactId>applib</artifactId>
      <version>${restfulobjects.version}</version>
    </dependency>
```

```
    <dependency>
      <groupId>org.starobjects.restful</groupId>
      <artifactId>viewer</artifactId>
      <version>${restfulobjects.version}</version>
    </dependency>
    ...
  </dependencies>
</dependencyManagement>
```

Note that an alternate approach for setting up dependencies is to have the parent module inherit from `org.starobjects.restful:release`. Doing it this way means that it isn't necessary to add entries to `<dependencyManagement>` because they are inherited. However, Maven2 does only allow a single parent, so this may not be an option for you if you want to use some other POM as your parent.

2.2. CommandLine Module

When prototyping we run the viewer from the command line (rather than deploying to a full webapp, a topic we cover in Chapter 5, *Deploying Restful Objects Webapps*). We therefore need to add a reference to the viewer in the `commandline` project's `pom.xml`:

```
<dependencies>
  ...
  <dependency>
    <groupId>org.starobjects.restful</groupId>
    <artifactId>viewer</artifactId>
    <version>${restfulobjects.version}</version>
  </dependency>
  ...
</dependencies>
```

2.3. Launch Configuration

To run using the Restul Viewer, we use the standard commandline bootstrapper `org.nakedobjects.runtime.NakedObjects` (as used for DnD and HTML viewer in prototype mode). Run with the arguments:

- --type exploration
- --viewer `org.starobjects.restful.viewer.embedded.RestfulViewerInstaller`

This sets up Jetty to run with the Restful Objects servlets and filters.

2.4. Testing the Viewer

Boot up Firefox and browse to http://localhost:8080. You should see links to access the services; these are (representations of) the registered services (typically repositories) in `nakedobjects.properties` configuration file.

> ### Note
>
> Firefox is currently the only supported web browser for testing in this way.

Chapter 3
Resources

Understanding how to interact with resources and how to interpret their representations is central to the RESTful approach. This chapter describes in detail the resources that are available; in effect, the API for using domain objects exposed using *Restful Objects*.

In order to invoke a HTTP method on a resource, the URL must be constructed correctly. After that, the client code then needs to handle the response.

Each of the resources is defined as interfaces annotated using `javax.ws.rs` (JAX-RS) annotations. JAX-RS is the Java API for Restful Web Services, part of Java EE 6. Its annotations are used by JAX-RS libraries such as RestEasy, or Jersey (the latter is the reference implementation). These libraries expose the resources as endpoints and route requests through to server-side methods (implemented by *Restful Objects*).

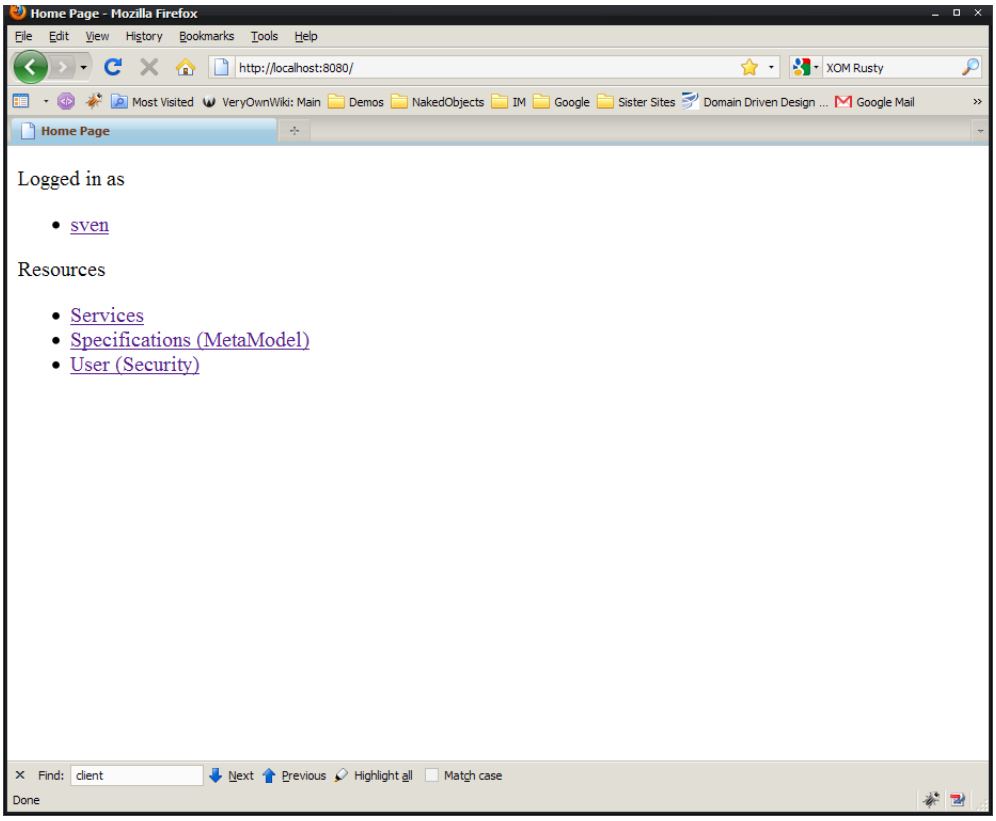
As such, we can use these interfaces as a way of describing the resources provided by *Restful Objects*. Let's go through each in turn.

3.1. HomePageResource

To start with, we have `HomePageResource`:

```
import javax.ws.rs.GET;
import javax.ws.rs.Produces;

public interface HomePageResource {

    @GET
    @Produces( {"application/xhtml+xml", "text/html"} )
    public String resources();
}
```

Because there is no `@javax.ws.rs.Path` annotation, this in effect says that the root URL "/" (eg http://localhost:8080/) is a valid resource. What you'll get back is an XHTML page that identifies the current user (in exploration mode this is always hard-coded), and then lists the resources available: the services (ie repositories), the specifications (metamodel), and details on the current user.



These blocks are always present on every page.

Meanwhile the raw XHTML - what your client code must parse - looks something like:

```
<?xml version="1.0"?>
<html>
  <head><title>Home Page</title></head>
  <body id="body">
    <div>
      <p>Logged in as</p>
      <ul class="nof-session">
        <li><a href="/user" rel="user" rev="resource" class="nof-user">sven</a></li>
      </ul>
    </div>
    <div class="nof-section">
      <p class="nof-section">Resources</p>
      <ul class="nof-resources">
        <li>
          <a href="/services" rel="services" rev="resources" class="nof-resource">
            Services
          </a>
        </li>
        <li>
          <a href="/specs" rel="specs" rev="resources" class="nof-resource">
            Specifications (MetaModel)
          </a>
        </li>
        <li>
          <a href="/user" rel="user" rev="resources" class="nof-resource">
            User (Security)
          </a>
        </li>
      </ul>
    </div>
  </body>
</html>
```

Note the use of `class` attributes to distinguish the different types of properties. For example, the XPath expression *//a[@class='nof-resource']/@href* will return the links to all resources available. One of these - */services* - is the list of services, so lets look at that next.

> **Note**
>
> Depending on feedback, the format of the XHTML may evolve in the future, eg to add in further class attributes.

3.2. Services Resource

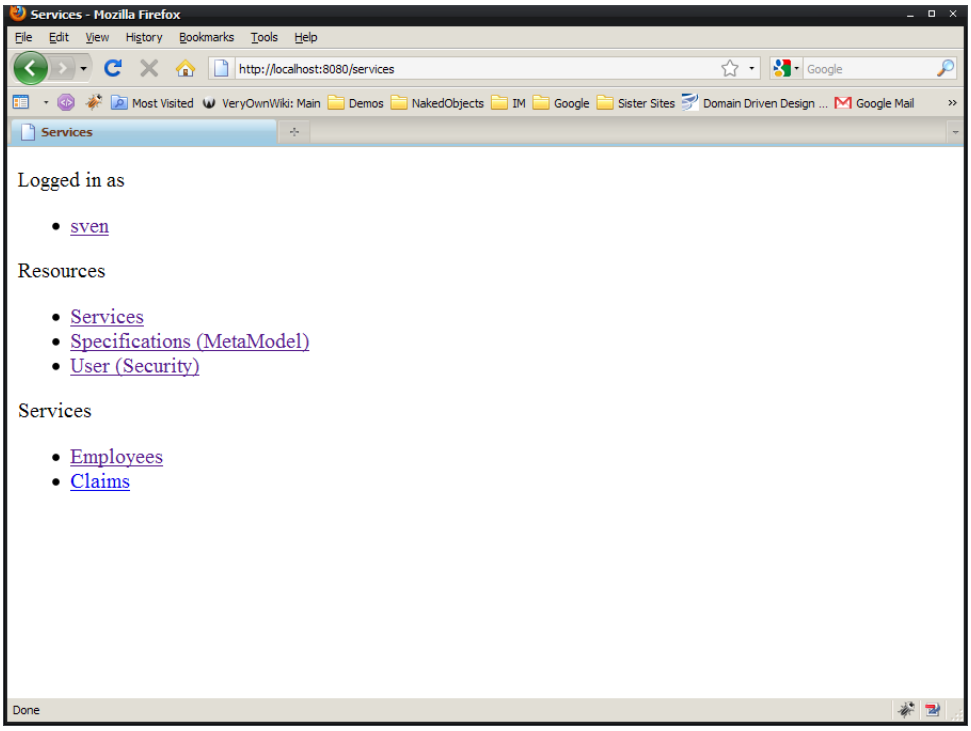
The */services* link indicated in Section 3.1, “HomePageResource” corresponds to the `ServicesResource`:

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

public interface ServicesResource {

  @GET
  @Produces( {"application/xhtml+xml", "text/html"} )
  @Path("/")
  public String services();
}
```

The implementation of this interface in *Restful Objects* viewer (`ServicesResourceImpl`) also defines a `@Path("/services")` for the class as a whole. This therefore defines a URL in the form */services* supporting the GET method.

> **Note**
>
> I believe that the `@Path("/services")` annotation should reside on the interface, not the implementation. This seems to be a limitation with RestEasy, the underlying library used by *Restful Objects*. Certainly for RestEasy 1.0.2 and also 1.1-rc2 this did not work, however.

Here's the resource that's returned, as shown in a browser:



The first two sections are the same; what's new is the list of services, corresponding to the registered services in `nakedobjects.properties`. The XHTML for this is:

```
<?xml version="1.0"?>
<html>
  <head><title>Services</title></head>
  <body id="body">
    ...
    <div class="nof-section">
      <p class="nof-section">Services</p>
      <ul class="nof-services">
        <li>
          <a href="/object/OID:1" rel="service" rev="services" class="nof-service">
            Employees
          </a>
        </li>
        <li>
          <a href="/object/OID:2" rel="service" rev="services" class="nof-service">
            Claims
          </a>
        </li>
      </ul>
    </div>
  </body>
</html>
```

Again, we can use XPath to pull back the resources:

- *//a[@class='nof-service']/@href* will return the hyperlinks to the object resources representing these services, in the form */object/OID*.

And let's look at object resources next.

3.3. Object Resource

The */object* link indicated above corresponds to the `ObjectResource` interface. This defines the set of interactions described in Section 1.2, “Introducing Restful Objects”:

```
import java.io.InputStream;

import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;

public interface ObjectResource {

  ...
}
```

The implementation of this interface in *Restful Objects* viewer (`ObjectResourceImpl`) also defines a `@Path("/object")` for the class as a whole. All URL paths are therefore prefixed */object/{oid}*.

> ### Note
>
> I believe that the `@Path` annotation should reside on the interface, not the implementation. This seems to be a limitation with RestEasy 1.0.2, the underlying library used by *Restful Objects*.

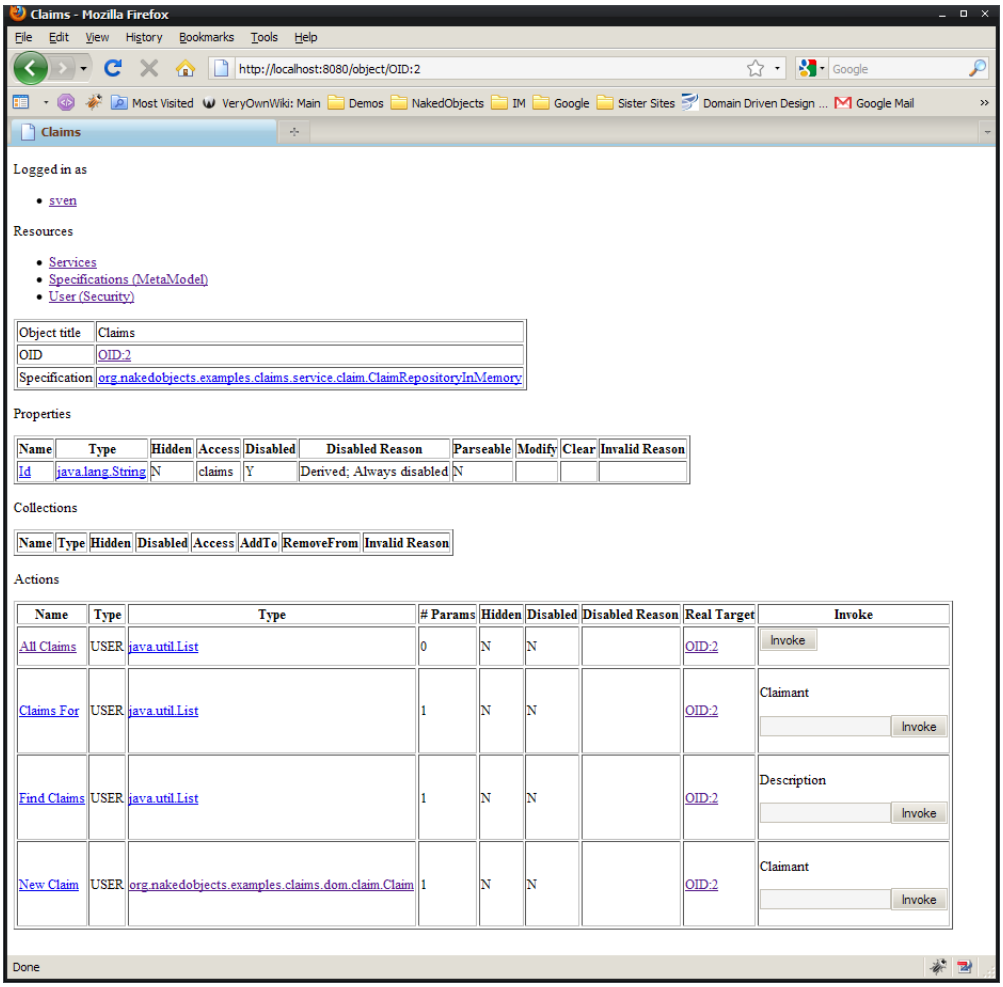
Let's break it down into sections, starting with the object.

Objects

The */objects/OID* resource corresponds to the `ObjectResource` interface:

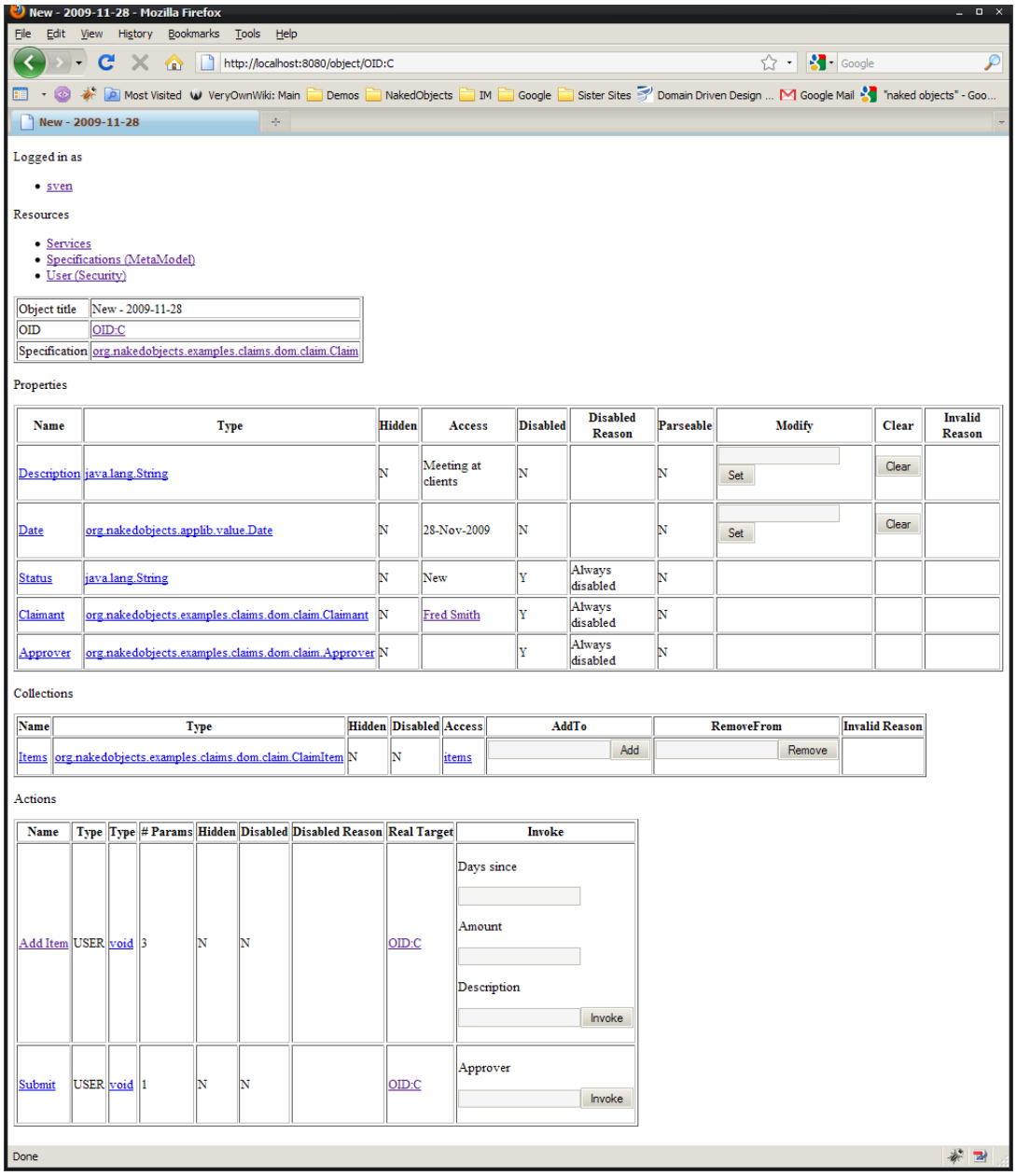
```
...

public interface ObjectResource {

  @GET
  @Path("/{oid}")
  @Produces( { "application/xhtml+xml", "text/html" })
  public String object(
    @PathParam("oid") final String oidStr);

  ...
}
```

This defines */object/{oid}* as a URL supporting GET. Here's what calling this method gives for a repository object:



As ever, we get the initial "logged in" and "resources" sections. The remaining sections are details about this object, providing its title, the properties, the list of collections and then the actions. For repositories, the only items of significance here are the actions.

Let's also look at a typical domain object:



The structure is the same, but here there are properties and collections, as well as actions. Let's look at the XHTML that represents this domain object, starting with the title:

Object title	New - 2009-11-28
OID	OID:C
Specification	org.nakedobjects.examples.claims.dom.claim.Claim

The raw *XHTML* looks like:

```
<div>
  <table border="1">
    <tr>
      <td>Object title</td>
      <td class="nof-title">
        <p>New - 2009-11-28</p>
     </td>
    </tr>
    <tr>
```

```
      <td>OID</td>
      <td class="nof-oid">
        <a href="/object/OID:C" rel="object" rev="object" class="nof-oid">
          OID:C
        </a>
      </td>
    </tr>
    <tr>
      <td>Specification</td>
      <td class="nof-specification">
        <a href="/specs/org.nakedobjects.examples.claims.dom.claim.Claim"
           rel="spec" rev="object" class="nof-specification">
          org.nakedobjects.examples.claims.dom.claim.Claim
        </a>
      </td>
    </tr>
  </table>
</div>
```

Some useful XPath expressions:

- *//td[@class='nof-title']/p/text()* will return the title
- *//td[@class='nof-oid']/a/text()* is the OID (as a string)
- *//td[@class='nof-specification']/a/@href* is a link the (resource repreenting the) specification of this object, in the form */specs/{fullyQualifiedClassName}*

Next up is the properties section:

Properties

Name	Type	Hidden	Access	Disabled	Disabled Reason	Parseable	Modify	Clear	Invalid Reason
Description	java.lang.String	N	Meeting at clients	N		N	Set	Clear	
Date	org.nakedobjects.applib.value.Date	N	28-Nov-2009	N		N	Set	Clear	
Status	java.lang.String	N	New	Y	Always disabled	N			
Claimant	org.nakedobjects.examples.claims.dom.claim.Claimant	N	Fred Smith	Y	Always disabled	N			
Approver	org.nakedobjects.examples.claims.dom.claim.Approver	N		Y	Always disabled	N			

The raw XHTML (abbreviated; just the first property is listed) is:

```
<div class="nof-properties">
  <p class="nof-properties">Properties</p>
  <table border="1">
    <tr>
      <th>Name</th>
      <th>Type</th>
      <th>Hidden</th>
      <th>Access</th>
      <th>Disabled</th>
      <th>Disabled Reason</th>
      <th>Parseable</th>
      <th>Modify</th>
      <th>Clear</th>
      <th>Invalid Reason</th>
    </tr>
    <tr>
```

```
      <td>
        <a href="/specs/org.nakedobjects.examples.claims.dom.claim.Claim/property/
description"
           rel="propertySpec" rev="property" class="nof-property">Description</a>
      </td>
      <td>
        <a href="/specs/java.lang.String" rel="propertyTypeSpec"
           rev="property" class="nof-property">java.lang.String</a>
      </td>
      <td>
        <p class="nof-visible">N</p>
      </td>
      <td>
        <p class="nof-property">Meeting at clients</p>
      </td>
      <td>
        <p class="nof-usable">N</p>
      </td>
      <td>
        <p />
      </td>
      <td>
        <p class="nof-visible">N</p>
      </td>
      <td>
        <div class="nof-property">
          <form name="property-description">
            <input type="value" name="proposedValue" />
            <input type="button" value="Set"
              onclick="modifyProperty(&quot;/object/
OID:C&quot;,&quot;description&quot;,proposedValue.value);" />
          </form>
        </div>
      </td>
      <td>
        <div class="nof-property">
          <form name="property-description">
            <input type="button" value="Clear"
              onclick="clearProperty(&quot;/object/OID:C&quot;,&quot;description&quot;);" />
          </form>
        </div>
      </td>
      <td>
        <p class="nof-valid" id="property-invalid-description" />
      </td>
    </tr>
    ...
  </table>
</div>
```

Some useful XPath expressions are:

- *//div[@class='nof-properties']//tr/td[1]/a/text()* returns the property names
- *//div[@class='nof-properties']//tr/td[2]/a/@href* returns links to (resources representing the) property types, in the form */specs/{fullyQualifiedClassName}*
- *//div[@class='nof-properties']//tr/td[3]/p/text()* returns whether properties are invisible (N=not invisible)
- *//div[@class='nof-properties']//tr/td[4]* returns the `<td>` table cells containing the values; the values will either be `<p>` (values) or `<a href=...>` (references)

and so on.

The "modify" and "clear" columns bear further description. These are used to perform PUTs and DELETEs on the resources that represent the property. Because XHTML4 does not support PUT and DELETE verbs in forms, they actually call Javascript fragments to do these calls. See the section called “Properties” for further details of the format of these requests.

The values that are entered into modify for value properties is the parseable text form (eg TRUE for a boolean); for reference properties it is the OID.

If a property is disabled, then the "disabled reason" column will indicate why. If a property is enabled but the proposed property value is invalid, then the "invalid reason" will indicate why. Note that it takes a round-trip to do this validation, because the domain objects are not in any sense serialized into the browser.

Let's now look at the collections section:

Collections

Name	Type	Hidden	Disabled	Access	AddTo	RemoveFrom	Invalid Reason
Items	org.nakedobjects.examples.claims.dom.claim.ClaimItem	N	N	items	Add	Remove	

The raw XHTML is:

```
<div class="nof-collections">
  <p class="nof-collections">Collections</p>
  <table border="1">
    <tr>
      <th>Name</th>
      <th>Type</th>
      <th>Hidden</th>
      <th>Disabled</th>
      <th>Access</th>
      <th>AddTo</th>
      <th>RemoveFrom</th>
      <th>Invalid Reason</th>
    </tr>
    <tr>
      <td>
        <a href="/specs/org.nakedobjects.examples.claims.dom.claim.Claim/collection/items"
           rel="propertySpec" rev="collection" class="nof-collection">
          Items
        </a>
      </td>
      <td>
        <a href="/specs/org.nakedobjects.examples.claims.dom.claim.ClaimItem"
           rel="propertyTypeSpec" rev="property" class="nof-property">
          org.nakedobjects.examples.claims.dom.claim.ClaimItem
        </a>
      </td>
      <td>
        <p class="nof-visible">N</p>
      </td>
      <td>
        <p class="nof-usable">N</p>
      </td>
      <td>
       <a href="/object/OID:C/collection/items"
          rel="collection" rev="nakedObject" class="nof-collection">items</a>
      </td>
      <td>
```

```
        <div class="nof-collection">
          <form name="collection-items">
            <input type="value" name="proposedValue" />
            <input type="button" value="Add"
              onclick="addToCollection(&quot;/object/OID:C&quot;, &quot;items&quot;,
 proposedValue.value);" />
          </form>
        </div>
      </td>
      <td>
        <div class="nof-collection">
          <form name="collection-items">
            <input type="value" name="proposedValue" />
            <input type="button" value="Remove"
              onclick="removeFromCollection(&quot;/object/OID:C&quot;, &quot;items&quot;,
 proposedValue.value);" />
          </form>
        </div>
      </td>
      <td>
        <p class="nof-valid" id="collection-invalid-items" />
      </td>
    </tr>
  </table>
</div>
```

In this particular case there is only a single collection, but in general there could be many.

Some useful XPath expressions are:

- *//div[@class='nof-collections']//tr/td[1]/a/text()* returns the collection names
- *//div[@class='nof-properties']//tr/td[2]/a/@href* returns links to (resources representing the) type of the collection, in the form */specs/{fullyQualifiedClassName}*
- *//div[@class='nof-properties']//tr/td[3]/p/text()* returns whether collections are invisible (N=not invisible)
- *//div[@class='nof-collections']//tr/td[5]/a/@href* returns a link to a resource representing the contents of the collection, in the form */object/{oid}/collection/{collectionId}*

and so on. Note that the table doesn't show the contents of each collection, instead it gives us a link to access the contents

Similar to properties, the "addTo" and "removeFrom" columns are used to submit PUT or DELETEs against the collection resource. See the section called “Collections” for more details. The arguments provided are the OIDs of objects in the collection.

Finally, let's look at actions:

Actions

Name	Type	Type	# Params	Hidden	Disabled	Disabled Reason	Real Target	Invoke
Add Item	USER	void	3	N	N		OID:C	Days since Amount Description Invoke
Submit	USER	void	1	N	N		OID:C	Approver Invoke

The raw XHTML (abbreviated; just the first action is listed) is:

```
<div class="nof-actions">
  <p class="nof-actions">Actions</p>
  <table border="1">
    <tr>
      <th>Name</th>
      <th>Type</th>
      <th>Type</th>
      <th># Params</th>
      <th>Hidden</th>
      <th>Disabled</th>
      <th>Disabled Reason</th>
      <th>Real Target</th>
      <th>Invoke</th>
    </tr>
    <tr>
      <td>
        <a href="/specs/org.nakedobjects.examples.claims.dom.claim.Claim/action/
addItem(int,double,java.lang.String)"
           rel="actionSpec" rev="action" class="nof-action">Add Item</a>
      </td>
      <td>
        <p class="nof-action">USER</p>
      </td>
      <td>
        <a href="/specs/void" rel="actionReturnTypeSpec" rev="action" class="nof-
action">void</a>
      </td>
      <td>
        <p class="nof-action">3</p>
      </td>
      <td>
        <p class="nof-visible">N</p>
      </td>
      <td>
        <p class="nof-usable">N</p>
      </td>
      <td>
        <p />
      </td>
      <td>
        <a href="/object/OID:C" rel="actionRealTarget" rev="action" class="nof-action">OID:C</
a>
      </td>
      <td>
        <div class="nof-action">
          <form name="action-addItem(int,double,java.lang.String)" method="POST"
```

```
                action="/object/OID:C/action/addItem(int,double,java.lang.String)">
            <p>Days since</p>
            <input type="value" name="arg0" />
            <p>Amount</p>
            <input type="value" name="arg1" />
            <p>Description</p>
            <input type="value" name="arg2" />
            <input type="submit" value="Invoke" />
          </form>
        </div>
      </td>
    </tr>
    ...
  </table>
</div>
```

Some useful XPath expressions are:

- *//div[@class='nof-actions']//tr/td[1]/a/text()* returns the names of each action
- *//div[@class='nof-actions']//tr/td[3]/a/@href* returns links to (resources representing the) type of the returned result of each action, in the form */specs/{fullyQualifiedClassName}*
- *//div[@class='nof-actions']//tr/td[5]/p/text()* returns whether each action is invisible or not (N=not invisible)
- *//div[@class='nof-actions']//tr/td[8]/a/text()* returns the OID of the real target; more on this in a moment
- *//div[@class='nof-actions']//tr/td[9]//form* returns a form to invoke the action using POST

and so on. Note that the table doesn't show the contents of each collection, instead it gives us a link to access the contents

There is no javascript calls this time; invoking actions is just a POST. The arguments provided are the same as for properties: text for parseable values, or the OIDs of objects for references.

The "realTarget" column is provided to support contributed actions. For example, here are the actions listed for a domain object (`Employee`) where the `claimsFor()` action is contributed (by the `ClaimsRepository`):

Object title	Tom Brown
OID	OID:4
Specification	org.nakedobjects.examples.claims.dom.employee.Employee

Properties ...

Collections ...

Actions

Name	Type	Type	# Params	Hidden	Disabled	Disabled Reason	Real Target	Invoke
Claims For	USER	java.util.List	1	N	N		OID:2	Claimant [Invoke]
New Claim	USER	org.nakedobjects.examples.claims.dom.claim.Claim	1	N	N		OID:2	Claimant [Invoke]

In the DnD viewer this action would appear to reside on the domain object. In reality though the action resides on the repository, taking a single argument - the domain object (`Employee`).

And that concludes our run-through of the GET verb for object resources. Phew!

Note

PUT and DELETE verbs for object resources have not yet been implemented.

Properties

The next set of resources exposed by `ObjectResource` are those specific to properties:

```
...
public interface ObjectResource {
  ...
  @PUT
  @Path("/{oid}/property/{propertyId}")
  @Produces( { "application/xhtml+xml", "text/html" })
  public String modifyProperty(
    @PathParam("oid") final String oidStr,
    @PathParam("propertyId") final String propertyId,
    @QueryParam("proposedValue") final String proposedValue);

  @DELETE
  @Path("/{oid}/property/{propertyId}")
  @Produces( { "application/xhtml+xml", "text/html" })
  public String clearProperty(
    @PathParam("oid") final String oidStr,
    @PathParam("propertyId") final String propertyId);
  ...
}
```

This defines */object/{oid}/property/{propertyId}* as the URL supporting:

- PUT, to change the value of a property; the proposedValue should be a query parameter to this URL;
- DELETE, to clear the property

Calling these will first validate the change, and if accepted apply the change:

- If the validation fails, then an exception will be thrown. This translates to a response with error code in the range [400, 500), and with a response header of "nof-reason".
- If the validation succeeds, the returned XHTML is the object's title only.

Collections

The next set of resources exposed by `ObjectResource` are those for collections:

```
...
public interface ObjectResource {
  ...
  @GET
  @Path("/{oid}/collection/{collectionId}")
  @Produces( { "application/xhtml+xml", "text/html" })
  public String accessCollection(
    @PathParam("oid") final String oidStr,
    @PathParam("collectionId") final String collectionId);

  @PUT
  @Path("/{oid}/collection/{collectionId}")
  @Produces( { "application/xhtml+xml", "text/html" })
  public String addToCollection(
    @PathParam("oid") final String oidStr,
    @PathParam("collectionId") final String collectionId,
    @QueryParam("proposedValue") final String proposedValueOidStr);
```

```
  @DELETE
  @Path("/{oid}/collection/{collectionId}")
  @Produces( { "application/xhtml+xml", "text/html" })
  public String removeFromCollection(
    @PathParam("oid") final String oidStr,
    @PathParam("collectionId") final String collectionId,
    @QueryParam("proposedValue") final String proposedValueOidStr);
  ...
}
```

This defines */object/{oid}/property/{collectionId}* as a URL supporting:

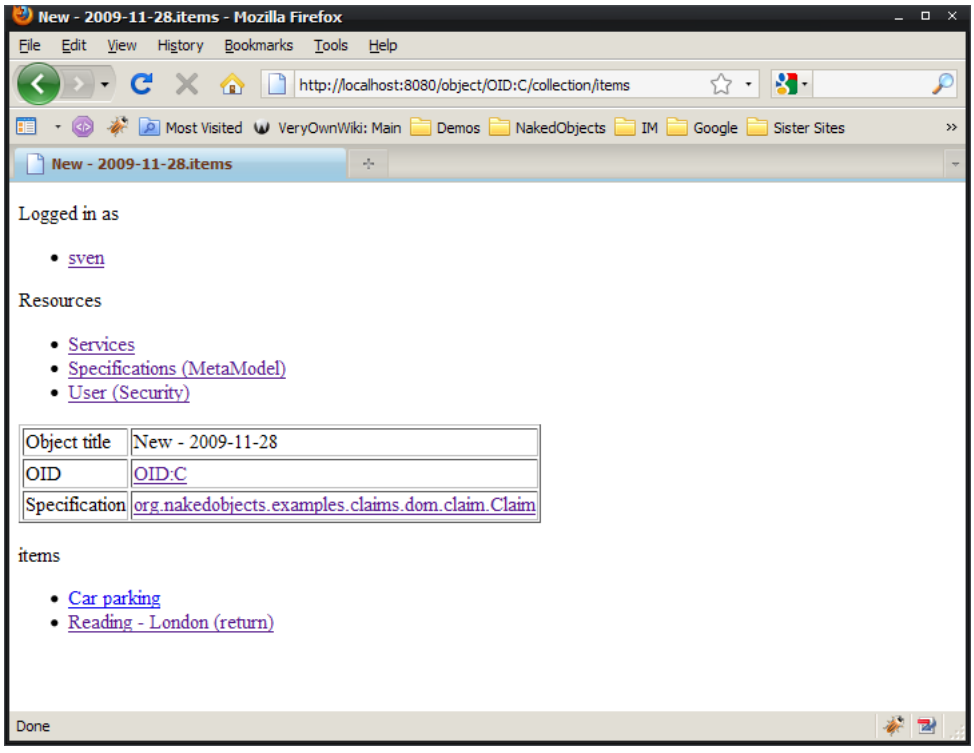
- GET, to read the contents of this collection.
- PUT, to add an item to the collection; the proposedValue should be a query parameter to this URL and contain an OID;
- DELETE, to remove an item to the collection; the proposedValue should be a query parameter to this URL and contain an OID;

The handling of PUT and the DELETE is similar to that of properties. The proposed value is validated first, and if invalid then a exception is thrown resulting in a response in range [400, 500) with a response header of "nof-reason". Otherwise the object's title and OID are returned.

Note

We could perhaps change this to return the new contents of the collection, ie as if a GET had been performed?

A GET meanwhile returns the following:



The raw XHTML for the contents part of this is:

```
<div class="nof-collection">
  <p class="nof-collection">items</p>
```

```
  <ul class="nof-collection">
    <li>
      <a href="/object/OID:D" rel="object" rev="results" class="nof-action-result">
        Car parking
      </a>
    </li>
    <li>
      <a href="/object/OID:E" rel="object" rev="results" class="nof-action-result">
        Reading - London (return)
      </a>
    </li>
  </ul>
</div>
```

Useful XPath here is:

- *//div[@class='nof-collection']//li/a/@href* to return the links to object resources in the collection

Actions

The final set of resources exposed by `ObjectResource` is for actions:

```
...
public interface ObjectResource {
  ...
  @POST
  @Path("/{oid}/action/{actionId}")
  @Produces( { "application/xhtml+xml", "text/html" })
  public String invokeAction(
    @PathParam("oid") final String oidStr,
    @PathParam("actionId") final String actionId,
    final InputStream body);
  ...
}
```

This defines */object/{oid}/property/{actionId}* as a URL supporting a POST. What's noteworthy here is the last argument, an `java.io.InputStream`. This provides a handle to the POST's input stream which contains the parameter/argument pairs.

The parameters should be named arg0, arg1, arg2 and so on, with the parameter value representing the argument. What this value is will depend on whether the parameter's type is a reference (entity) type or a parseable value type:

- for reference types, the value should be the (string representation of the) OID of the entity
- for value types, the value should be in parseable string format. In practical terms, use the same string format as would work in the the DnD viewer. (To be precise, it's the format understood by the `ParseableFacet` for the parameter's type)

For example, for an action findCustomers(registeredDate, CustomerType) then the parameters would be something like:

- arg0=20090103
- arg1=CTP|12

where **`20090103`** is 3-Jan-2009 in parseable format, and **`CTP|12`** is the OID (as assigned by the object store) for `CustomerType` with ID=12.

When an action is invoked, the response always includes the OID and title of the object on which the action was invoked. In addition:

- if the action returned an object, then the response will include a link to the object

> **Note**
>
> We could perhaps change this to return the GET of the returned object?

- if the action returned a collection of objects, then the response will return a list of links to the objects
- if the action returned void; then nothing further is added to the response

3.4. Metamodel (specs) Resource

The */specs/{fullyQualifedClassName}* links indicated in several places in the representations produced by `ObjectResource` corresponds to the `SpecsResource`:

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;

public interface SpecsResource {
  ...
}
```

The implementation of this interface in *Restful Objects* viewer (`SpecsResourceImpl`) also defines a `@Path("/specs")` for the class as a whole. This therefore defines a URL in the form */services* supporting the GET method.

> **Note**
>
> I believe that the `@Path` annotation should reside on the interface, not the implementation. This seems to be a limitation with RestEasy 1.0.2, the underlying library used by *Restful Objects*.

The purpose of the /specs/ family of resources is to describe the structure of the domain objects, ie expose a metamodel for the domain objects. Client-side applications might choose to iterate through all the specs resources first and cache them; this would then simplify the task of rendering domain objects.

Again, let's break the resources provided by `SpecsResource` into sections.

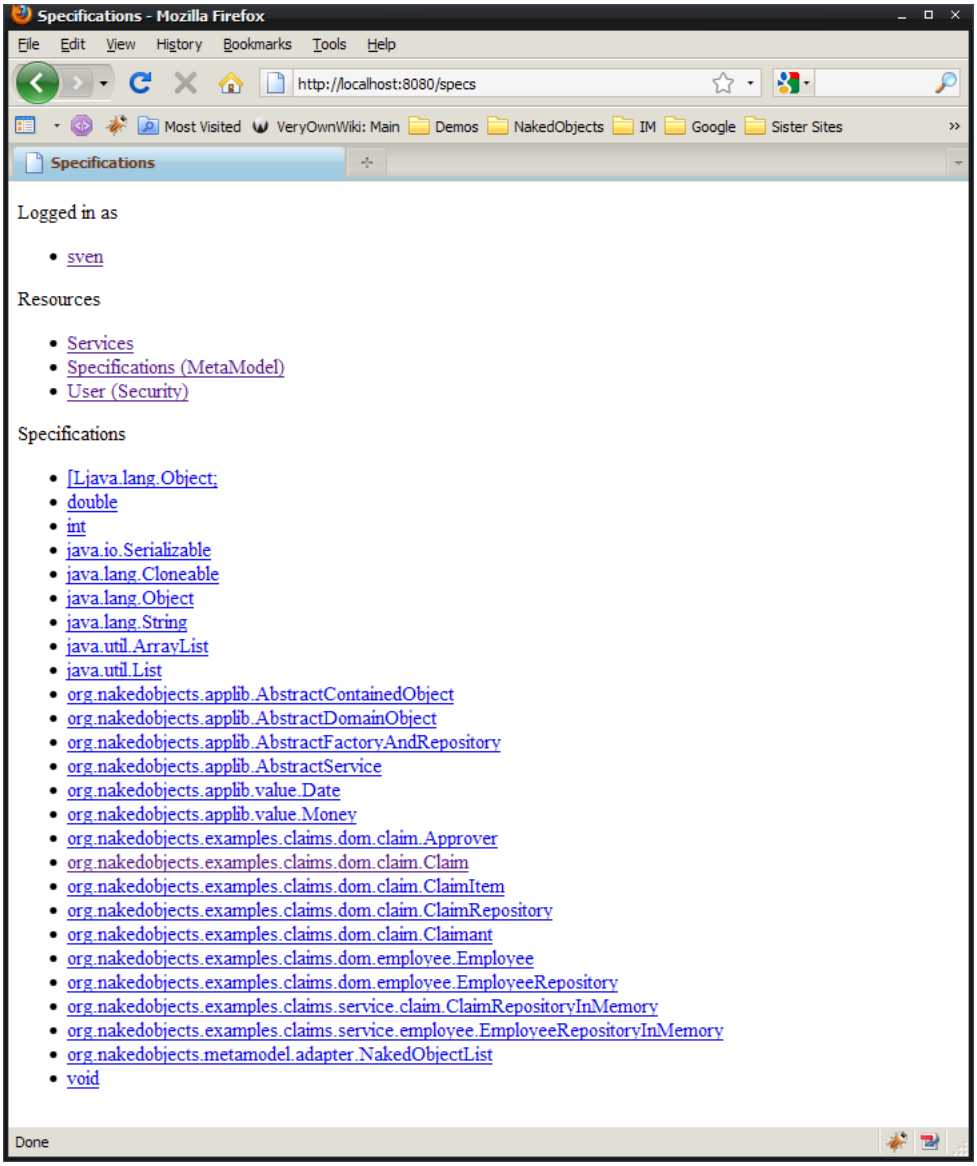
All Classes

First up, we have a resource to list all classes (or specs):

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;

public interface SpecsResource {

  @GET
  @Path("/")
  @Produces( { "application/xhtml+xml", "text/html" })
  public abstract String specs();
```

```
  ...
}
```

This defines */specs* as a URL accepting GET. This returns the following:



The raw XHTML produced (abbreviated) is:

```
<div class="nof-section">
  <p class="nof-section">Specifications</p>
  <ul class="nof-specifications">
    ...
    <li>
      <a href="/specs/int" rel="spec" rev="specs"
         class="nof-specification">int</a>
    </li>
    ...
    <li>
      <a href="/specs/java.lang.String" rel="spec" rev="specs"
         class="nof-specification">java.lang.String</a>
    </li>
    ...
    <li>
      <a href="/specs/org.nakedobjects.applib.value.Date" rel="spec" rev="specs"
         class="nof-specification">org.nakedobjects.applib.value.Date</a>
```

```
    </li>
    ..
    <li>
      <a href="/specs/org.nakedobjects.examples.claims.dom.claim.Claim" rel="spec"
 rev="specs"
         class="nof-specification">org.nakedobjects.examples.claims.dom.claim.Claim</a>
    </li>
    <li>
      <a href="/specs/org.nakedobjects.examples.claims.dom.claim.ClaimItem" rel="spec"
 rev="specs"
         class="nof-specification">org.nakedobjects.examples.claims.dom.claim.ClaimItem</a>
    </li>
    ...
    <li>
      <a href="/specs/void" rel="spec" rev="specs"
         class="nof-specification">void</a>
    </li>
  </ul>
</div>
```

Useful XPath expressions:

- *//a[@class='nof-specification']/@href* will give links to resources for all specifications

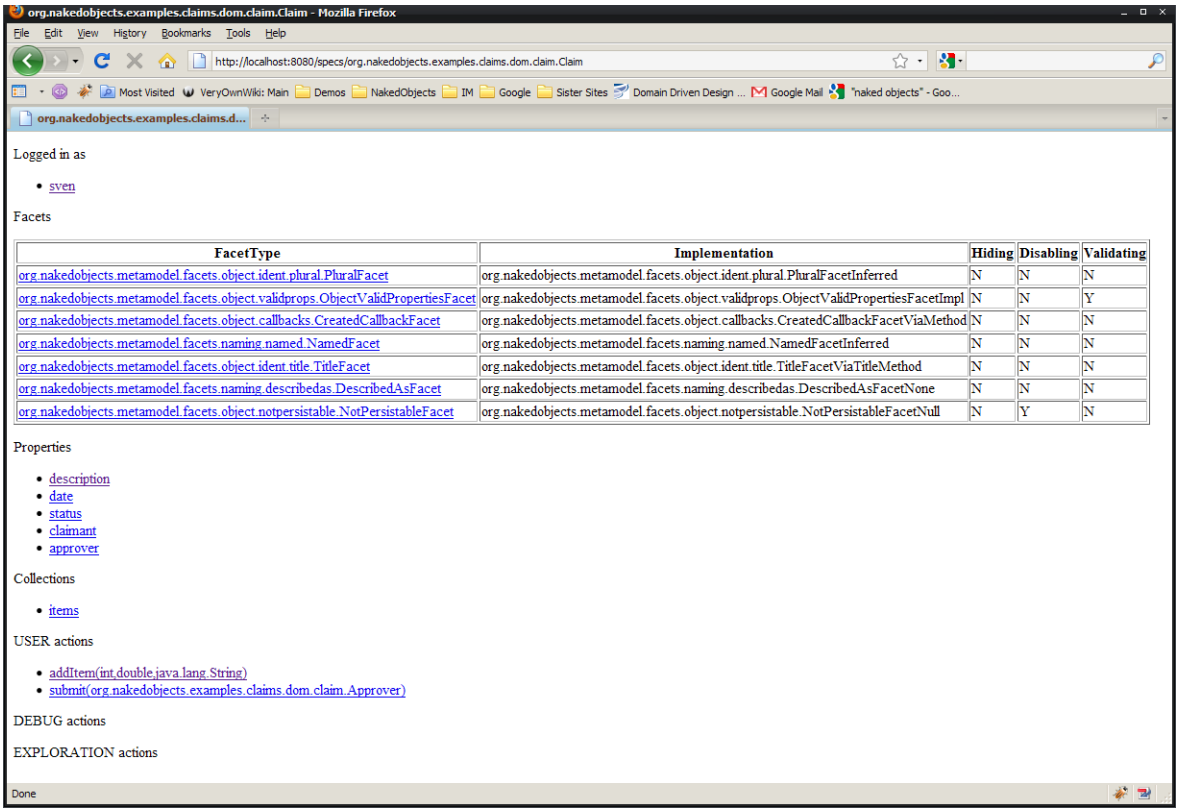
Let's look at an individual specification next.

Class (`NakedObjectSpecification`)

The resource for a specification is:

```
...

public interface SpecsResource {
  ...
  @GET
  @Path("/{specFullName}")
  @Produces( { "application/xhtml+xml", "text/html" })
  public abstract String spec(
    @PathParam("specFullName") final String specFullName);
  ...
}
```

This defines */specs/{specFullName}* as a URL accepting GET. This returns:



The raw XHTML for this breaks into five regions.

First (abbreviated) we have the facets, which define additional semantics to the holder (in this case, the spec):

```
<div class="nof-facets">
  <p class="nof-facets">Facets</p>
  <table border="1">
     <tr>
       <th>FacetType</th>
       <th>Implementation</th>
       ...
     </tr>
     ...
     <tr>
       <td>
         <a href="org.nakedobjects.examples.claims.dom.claim.Claim/facet/
org.nakedobjects.metamodel.facets.object.ident.plural.PluralFacet"
            rel="facet" rev="spec" class="nof-
facet">org.nakedobjects.metamodel.facets.object.ident.plural.PluralFacet</a>
       </td>
       <td>
         <p>org.nakedobjects.metamodel.facets.object.ident.plural.PluralFacetInferred</p>
       </td>
       ...
     </tr>
     ...
     <tr>
       <td>
         <a href="org.nakedobjects.examples.claims.dom.claim.Claim/facet/
org.nakedobjects.metamodel.facets.naming.named.NamedFacet"
            rel="facet" rev="spec" class="nof-
facet">org.nakedobjects.metamodel.facets.naming.named.NamedFacet</a>
       </td>
       <td>
         <p>org.nakedobjects.metamodel.facets.naming.named.NamedFacetInferred</p>
```

```
      </td>
      ...
    </tr>
    ...
    <tr>
      <td>
        <a href="org.nakedobjects.examples.claims.dom.claim.Claim/facet/
org.nakedobjects.metamodel.facets.naming.describedas.DescribedAsFacet" rel="facet" rev="spec"
 class="nof-facet">org.nakedobjects.metamodel.facets.naming.describedas.DescribedAsFacet</a>
      </td>
      <td>
        <p>org.nakedobjects.metamodel.facets.naming.describedas.DescribedAsFacetNone</p>
      </td>
      ...
    </tr>
    ...
  </table>
</div>
```

Many of the facets listed will not be that relevant to us, but some - such as the singular name of a class (`NamedFacet`), the plural name of a class (`PluralFacet`) and the description of a class (`DescribedAsFacet`) will be useful for presentation purposes. It is also possible to define additional facets that might be relevant to your own client-side application. For example, if you were writing a mash-up GUI and wanted to render an Address domain object within a map, you might want to define a `MapCoordinatesFacet`.

> ## Note
>
> Actually, this isn't quite true; there is currently no way to evaluate a facet for a particular domain object instance.

As ever, we can use XPath to pull out values:

- *//div[@class='nof-facets']//a[.='org.nakedobjects.metamodel.facets.object.ident.plural.PluralFacet']/@href* will pull out the link for the `PluralFacet`, if there is one.

Next we have the (definition of the) properties for a spec:

```
<div class="nof-properties">
  <p class="nof-properties">Properties</p>
  <ul class="nof-properties">
    <li>
      <a href="/specs/org.nakedobjects.examples.claims.dom.claim.Claim/property/description"
         rel="property" rev="spec" class="nof-property">description</a>
    </li>
    <li>
      <a href="/specs/org.nakedobjects.examples.claims.dom.claim.Claim/property/date"
         rel="property" rev="spec" class="nof-property">date</a>
   </li>
   ...
  </ul>
</div>
```

Useful XPath queries here:

- *//div[@class='nof-properties']//a/text()* returns the property names
- *//div[@class='nof-properties']//a/@href* returns links to the property definitions (see the section called “Class Members (`NakedObjectMember`)”)

Similarly, we have collections:

```
<div class="nof-collections">
  <p class="nof-collections">Collections</p>
  <ul class="nof-collections">
    <li>
      <a href="/specs/org.nakedobjects.examples.claims.dom.claim.Claim/collection/items"
         rel="collection" rev="spec" class="nof-collection">items</a>
    </li>
    ...
  </ul>
</div>
```

Useful XPath queries here:

- *//div[@class='nof-collections']//a/text()* returns the collection names
- *//div[@class='nof-collections']//a/@href* returns links to the collection definitions (see the section called “Class Members (`NakedObjectMember`)”)

Lastly, the actions. These fall into three groups: regular (USER) actions, debug actions (annotated with `@Debug`) and exploration actions (that are available only in exploration mode):

```
<div class="nof-actions">
  <p class="nof-actions">USER actions</p>
  <ul class="nof-actions">
    <li>
      <a href="/specs/org.nakedobjects.examples.claims.dom.claim.Claim/action/
addItem(int,double,java.lang.String)"
         rel="action" rev="spec" class="nof-action">addItem(int,double,java.lang.String)</a>
    </li>
    <li>
      <a href="/specs/org.nakedobjects.examples.claims.dom.claim.Claim/action/
submit(org.nakedobjects.examples.claims.dom.claim.Approver)"
         rel="action" rev="spec" class="nof-
action">submit(org.nakedobjects.examples.claims.dom.claim.Approver)</a>
    </li>
  </ul>
</div>
<div class="nof-actions">
  <p class="nof-actions">DEBUG actions</p>
  <ul class="nof-actions" />
</div>
<div class="nof-actions">
  <p class="nof-actions">EXPLORATION actions</p>
  <ul class="nof-actions" />
</div>
```

Useful XPath queries here:

- *//div[@class='nof-actions' and p/text()='USER actions']//a/@href* returns links to the regular user actions
- *//div[@class='nof-actions' and p/text()='DEBUG actions']//a/@href* returns links to the debug actions
- *//div[@class='nof-actions' and p/text()='EXPLORATION actions']//a/@href* returns links to the exploration actions

Class Members (`NakedObjectMember`)

The next set of resources provided by SpecsResource are for the individual class members (properties, collections or actions):

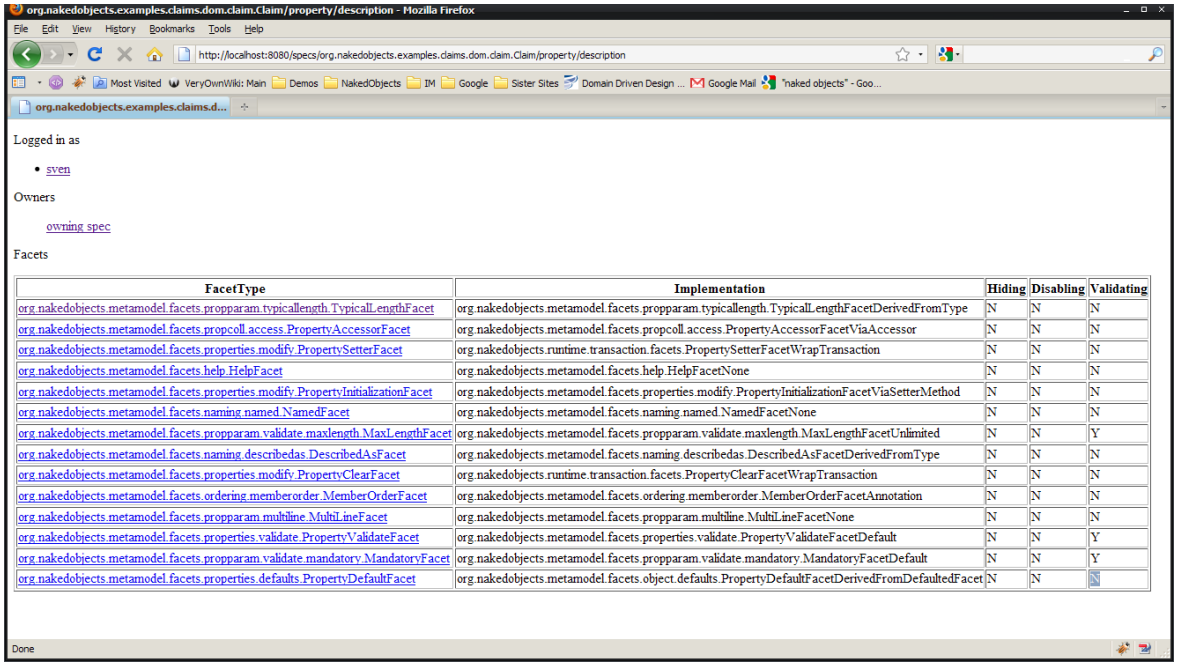
```
...

public interface SpecsResource {
```

```
  ...
  @GET
  @Path("/{specFullName}/property/{propertyName}")
  @Produces( { "application/xhtml+xml", "text/html" })
  public abstract String specProperty(
    @PathParam("specFullName") final String specFullName,
    @PathParam("propertyName") final String propertyName);

  @GET
  @Path("/{specFullName}/collection/{collectionName}")
  @Produces( { "application/xhtml+xml", "text/html" })
  public abstract String specCollection(
    @PathParam("specFullName") final String specFullName,
    @PathParam("collectionName") final String collectionName);

  @GET
  @Path("/{specFullName}/action/{actionId}")
  @Produces( { "application/xhtml+xml", "text/html" })
  public abstract String specAction(
    @PathParam("specFullName") final String specFullName,
    @PathParam("actionId") final String actionId);
  ...
}
```

This defines the following URLs all accepting GET:

- */specs/{specFullName}/property/{propertyName}* for a resource representing a property definition
- */specs/{specFullName}/collection/{propertyName}* for a resource representing a collection definition
- */specs/{specFullName}/action/{actionId}* for a resource representing a action definition

Each of these resources generates a similar representation, listing the facets for that class member. For example, here is the resource for a property spec:



The raw XHTML (abbreviated) is:

```
<div>
  <p>Owners</p>
  <ul class="nof-specification">
    <a href="/specs/org.nakedobjects.examples.claims.dom.claim.Claim" rel="spec"
 rev="property" class="nof-specification">owning spec</a>
```

```
  </ul>
</div>
<div class="nof-facets">
  <p class="nof-facets">Facets</p>
  <table border="1">
    <tr>
      <th>FacetType</th>
      <th>Implementation</th>
      ...
    </tr>
    <tr>
      <td>
        <a href="description/facet/
org.nakedobjects.metamodel.facets.propparam.typicallength.TypicalLengthFacet" rel="facet"
 rev="spec" class="nof-
facet">org.nakedobjects.metamodel.facets.propparam.typicallength.TypicalLengthFacet</a>
      </td>
      <td>

 <p>org.nakedobjects.metamodel.facets.propparam.typicallength.TypicalLengthFacetDerivedFromType</
p>
      </td>
      ...
    </tr>
    ...
    <tr>
      <td>
        <a href="description/facet/
org.nakedobjects.metamodel.facets.ordering.memberorder.MemberOrderFacet" rel="facet"
 rev="spec" class="nof-
facet">org.nakedobjects.metamodel.facets.ordering.memberorder.MemberOrderFacet</a>
      </td>
      <td>
        <p>org.nakedobjects.metamodel.facets.ordering.memberorder.MemberOrderFacetAnnotation</
p>
      </td>
      ...
    </tr>
    <tr>
      <td>
        <a href="description/facet/
org.nakedobjects.metamodel.facets.propparam.validate.mandatory.MandatoryFacet" rel="facet"
 rev="spec" class="nof-
facet">org.nakedobjects.metamodel.facets.propparam.validate.mandatory.MandatoryFacet</a>
      </td>
      <td>

 <p>org.nakedobjects.metamodel.facets.propparam.validate.mandatory.MandatoryFacetDefault</p>
      </td>
      ...
    </tr>
    ...
  </table>
</div>
```

As for specifications themselves (see the section called “Class (`NakedObjectSpecification`)”), many of the facets will not be that relevant. However, `NamedFacet` and `DescribedAsFacet` mentioned earlier would be, for presentation purposes. In addition, for properties the `TypicalLengthFacet` can be used as a hint for a field in the UI, and `MemberOrderFacet` annotation can be used to indicate the order of fields in the UI. The `MandatoryFacet` can be used to indicate mandatory properties.

As ever, XPath can be used to pull out information from the resource.

Note

Although actions produce a similar output, they ought to be extended to provide information on action parameters; both how many parameters there are, and also facets associated with those parameters.

Facets

A facet resource allows the value of a facet to be inspected. The resources provided by `SpecsResource` are:

```
...

public interface SpecsResource {
  ...
  @GET
  @Path("/{specFullName}/facet/{facetType}")
  @Produces( { "application/xhtml+xml", "text/html" })
  public abstract String specFacet(
    @PathParam("specFullName") final String specFullName,
    @PathParam("facetType") final String facetTypeName);

  @GET
  @Path("/{specFullName}/property/{propertyName}/facet/{facetType}")
  @Produces( { "application/xhtml+xml", "text/html" })
  public abstract String specPropertyFacet(
    @PathParam("specFullName") final String specFullName,
    @PathParam("propertyName") final String propertyName,
    @PathParam("facetType") final String facetTypeName);

  @GET
  @Path("/{specFullName}/collection/{collectionName}/facet/{facetType}")
  @Produces( { "application/xhtml+xml", "text/html" })
  public abstract String specCollectionFacet(
    @PathParam("specFullName") final String specFullName,
    @PathParam("collectionName") final String collectionName,
    @PathParam("facetType") final String facetTypeName);

  @GET
  @Path("/{specFullName}/action/{actionId}/facet/{facetType}")
  @Produces( { "application/xhtml+xml", "text/html" })
  public abstract String specActionFacet(
    @PathParam("specFullName") final String specFullName,
    @PathParam("actionId") final String actionId,
    @PathParam("facetType") final String facetTypeName);
  ...
}
```

This defines the following URLs all accepting GET:

- */specs/{specFullName}/facet/{facetType}* for a facet on a spec
- */specs/{specFullName}/property/{propertyName}/facet/{facetType}* for a facet on a property
- */specs/{specFullName}/collection/{propertyName}/facet/{facetType}* for a facet on a collection
- */specs/{specFullName}/action/{actionId}/facet/{facetType}* for a facet on an action

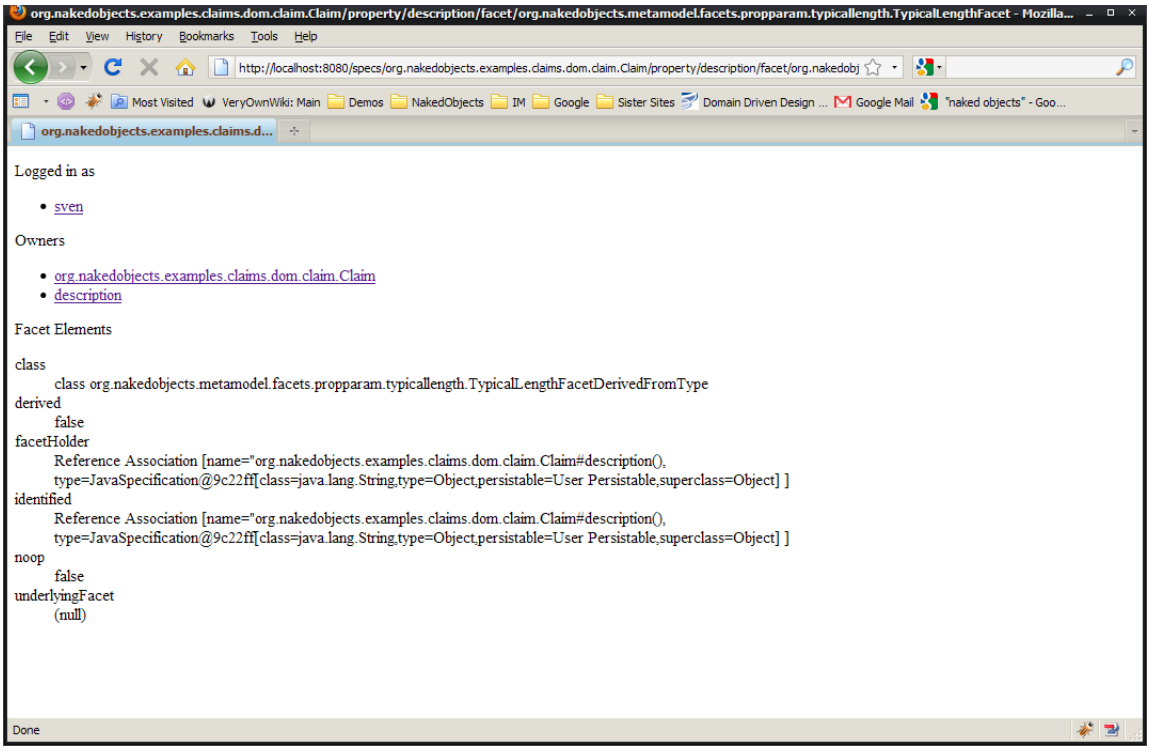
Note

In addition we need a resource to allow the facets of an actions parameter to be queried.

Note

(Mentioned elsewhere), we also need to provide the ability to evaluate facets per instance. Although many facets are per class, some (such as `TitleFacet`) will vary by instance. This will (presumably) need some additional resource methods (eg `specActionFacetFor(...)`) that take an oid as a parameter.

Each of these resources generates a similar representation, evaluating a facets for its facet holder. For example, here is the resource for a property spec:



The raw XHTML (abbreviated) is:

```
<div>
  <p>Owners</p>
  <ul class="nof-properties">
    <li>
      <a href="/specs/org.nakedobjects.examples.claims.dom.claim.Claim" rel="owning spec"
         rev="spec" class="facet">
        org.nakedobjects.examples.claims.dom.claim.Claim
      </a>
    </li>
    <li>
      <a href="/specs/org.nakedobjects.examples.claims.dom.claim.Claim/property/description"
         rel="owning property" rev="property" class="facet">
        description
      </a>
    </li>
  </ul>
</div>
<div class="nof-facet-elements">
  <p class="nof-facet-elements">Facet Elements</p>
  <dl class="nof-facet-elements">
    <dt>class</dt>
    <dd>class
 org.nakedobjects.metamodel.facets.propparam.typicallength.TypicalLengthFacetDerivedFromType</
dd>
```

```
    <dt>derived</dt>
    <dd>false</dd>
    <dt>facetHolder</dt>
    <dd>Reference Association
 [name="org.nakedobjects.examples.claims.dom.claim.Claim#description(),
 type=JavaSpecification@9c22ff[class=java.lang.String,type=Object,persistable=User
 Persistable,superclass=Object] ]</dd>
    <dt>identified</dt>
    <dd>Reference Association
 [name="org.nakedobjects.examples.claims.dom.claim.Claim#description(),
 type=JavaSpecification@9c22ff[class=java.lang.String,type=Object,persistable=User
 Persistable,superclass=Object] ]</dd>
    <dt>noop</dt>
    <dd>false</dd>
    <dt>underlyingFacet</dt>
    <dd>(null)</dd>
  </dl>
</div>
```

This representation is created by applying a JavaBean conventions on the facet (all `getXxx()` and `isXxx()` methods exposed by the facet itself).

> **Note**
>
> We need a more general purpose mechanism to query facets. As can be seen, the above doesn't actually expose the typical value of the facet, because the method we want is called `value()`, not `getValue()`.

Some XPath queries that might be useful are:

- *//div[@class='nof-facet-elements']//dt[.='class']/following-sibling::dd[1]* will return the `<dd>` element corresponding to the 'class' `<dt>`
- *//div[@class='nof-facet-elements']//dt[.='derived']/following-sibling::dd[1]* will return the `<dd>` element corresponding to the 'derived' `<dt>d`

3.5. Security (user) Resource

To finish up, the */user* link indicated in Section 3.1, “HomePageResource” corresponds to the `UserResource`:

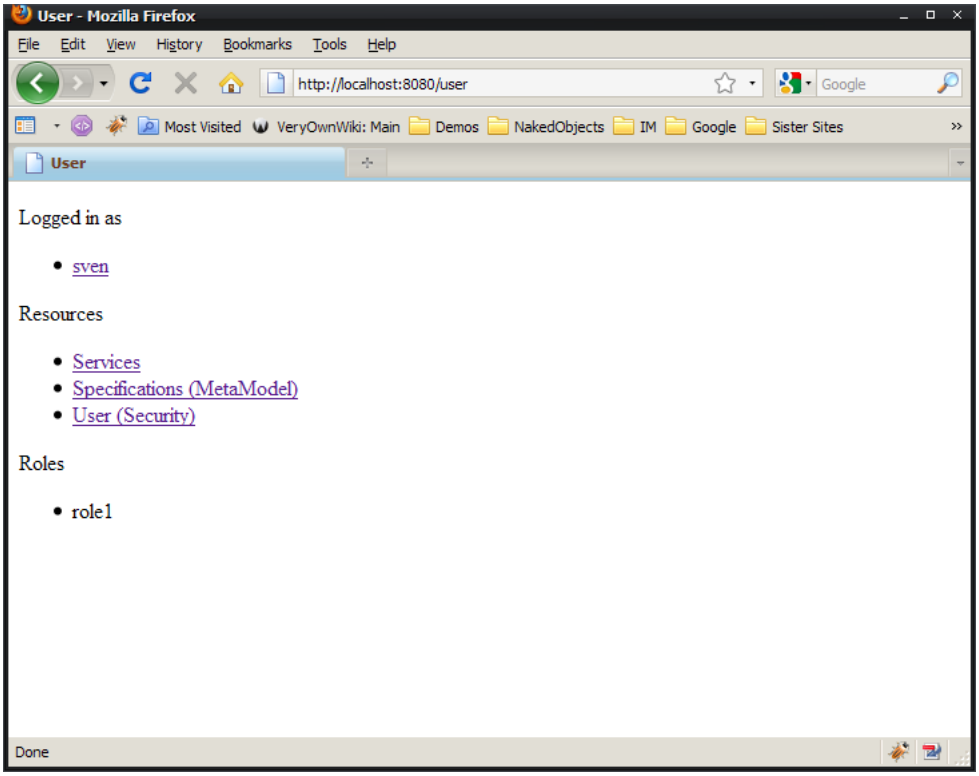
```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

public interface UserResource {

  @GET
  @Produces( { "application/xhtml+xml", "text/html" })
  public String user();
}
```

The implementation of this interface in *Restful Objects* viewer (`UserResourceImpl`) also defines a `@Path("/user")` for the class as a whole. Taken together this therefore defines a URL in the form */user* supporting the GET method.

Note

I believe that the `@Path("/user")` annotation should reside on the interface, not the implementation. This seems to be a limitation with RestEasy 1.0.2, the underlying library used by *Restful Objects*.

Here's the resource that's returned, as shown in a browser:



The first section (as ever) lists the current user, while the final section lists out the roles for the current user. The XHTML for this is:

```
<?xml version="1.0"?>
<html>
  <head><title>User</title></head>
  <body id="body">
    <div>
      <p>Logged in as</p>
      <ul class="nof-session">
        <li>
          <a href="/user" rel="user" rev="resource" class="nof-user">sven</a>
        </li>
      </ul>
    </div>
    ...
    <div class="nof-section">
      <p class="nof-section">Roles</p>
      <ul class="nof-roles">
        <li>
          <p class="nof-role">role1</p>
        </li>
      </ul>
    </div>
  </body>
</html>
```

Again, we can use XPath to pull back the resources:

- *//a[@class="nof-user"]/text()* returns the current user name
- *//p[@class="nof-role"]/text()* returns the role names

Chapter 4

Writing Client-side Applications

The point of *Restful Objects* is to provide our domain objects as RESTful resources so that they can be used by any other client application. These applications can be written in any language; so long as they can submit HTTP requests and can parse XHTML, they can interact with the resources provided by *Restful Objects*.

That said, if you are writing Java applications, then *Restful Objects* provides an application library (applib) to simplify the task. To reference this applib, add the following to your Maven pom's `<dependencies>` section:

```
<properties>
  <restfulobjects.version>1.0.0</restfulobjects.version>   <!-- OR WHATEVER -->
</properties>

<dependencies>
  ...
  <dependency>
    <groupId>org.starobjects.restful</groupId>
    <artifactId>applib</artifactId>
    <version>${restfulobjects.version}</version>
  </dependency>
  ...
</dependencies>
```

You then have a choice of approaches.

4.1. AbstractRestfulClient

To get you started you might want to use the adapter, `AbstractRestfulClient`, available in the *Restful Objects*' applib. This exposes some of the HTTP methods, including GET and POST, and serves up resources as XML documents (using Elliot Rusty Harold's XOM library). There is also a `XomUtils` class, that provides some pretty-printing support and a couple of other helper methods.

Note

`AbstractRestfulClient` is not particularly comprehensive (at the time of writing it doesn't include support for PUT or DELETE, for example). It may well gain extra methods in future releases, however.

4.2. RestEasy's Client-side Framework

Alternatively, you might want to look into JBoss RestEasy, which provides a client-side framework to eliminate some of the boilerplate. This uses the resource interfaces (see Chapter 3, *Resources*) to create client-side stubs.

Note

Due to a fact that `@Path` is annotated on the implementations - not the interfaces - of `ObjectResource`, `ServiceResource`, `SpecsResource` and `UserResource`, it's possible that this approach will not work until we upgrade the version of RestEasy.

Chapter 5

Deploying Restful Objects Webapps

At some point you'll presumably want to deploy your domain object using *Restful Objects* as a webapp.

If you've used the Naked Objects archetype then it will have created a webapp project for you, with its `web.xml` set up with the servlets and filters for the HTML viewer. We can use this as the basis for *Restful Objects*' configuration. (In principle, the `web.xml` could support both the HTML viewer and *Restful Objects*. This chapter assumes you only want to expose the latter interface, however).

In addition, you'll need to decide on the authentication mechanism. *Restful Objects* comes with some out-of-the-box support, but you may want to extend it for your own purposes. This chapter shows you how.d

5.1. Update POM Dependencies

First up (just as we did when prototyping, see Chapter 2, *Using Restful Objects in Prototypes*), we need to reference the *Restful Objects* viewer:

```
<dependencies>
  ...
  <dependency>
    <groupId>org.starobjects.restful</groupId>
    <artifactId>viewer</artifactId>
  </dependency>
  ...
</dependencies>
```

Here I'm assuming that this is in the webapp submodule where the parent defines the version in its `<dependencyManagement>` section (see Section 2.1, “Parent Module”).

5.2. web.xml

Next, we need to update `web.xml` to specify the servlets, listeners and filters needed for *Restful Objects*. You'll find that some of these are shared with HTML viewer.

DTD and Display Name

To start with, we have the boilerplate DTD reference and `<displayname>`:t

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN" "http://
java.sun.com/dtd/web-app_2_3.dtd"[]>
<web-app>
  <display-name>Restful Objects</display-name>

  ...
</web-app>
```

Context Parameters

There is one context parameter to add:

```
<web-app>
  ...
  <context-param>
    <param-name>javax.ws.rs.Application</param-name>
    <param-value>org.starobjects.restful.viewer.RestfulApplication</param-value>
  </context-param>
  ...
</web-app>
```

This is used by RestEasy to defines the supported resources.

Filters and Filter Mappings

Next, filters.

```
<web-app>
  ...
  <filter>
    <filter-name>NakedObjectsSessionFilter</filter-name>
    <filter-class>org.nakedobjects.webapp.NakedObjectsSessionFilter</filter-class>
    <init-param>
      <param-name>authenticationSessionLookupStrategy</param-name>
      <param-
value>org.starobjects.restful.viewer.authentication.AuthenticationSessionLookupStrategyExtended</
param-value>
    </init-param>
  </filter>

  <filter>
    <filter-name>NakedObjectsStaticContentFilter</filter-name>
    <filter-class>org.nakedobjects.webapp.NakedObjectsSessionFilter</filter-class>
  </filter>
  ...
</web-app>
```

The `NakedObjectsSessionFilter` is used to ensure that there is an authentication session in place. As you might infer from its `<init-param>`, the mechanism it does to lookup this authentication session is pluggable. The one specified will fake a session if running in EXPLORATION mode. But, again, more on this in Section 5.4, “Authentication”.

The `NakedObjectsStaticContentFilter` is used to decorate any static resources (Javascript, CSS or images) so that they are cached client-side.

The mappings for these two filters are:

```
<web-app>
  ...
  <filter-mapping>
    <filter-name>NakedObjectsSessionFilter</filter-name>
    <url-pattern>*</url-pattern>
  </filter-mapping>

  <filter-mapping>
    <filter-name>NakedObjectsStaticContentFilter</filter-name>
    <url-pattern>*.js</url-pattern>
  </filter-mapping>
  <filter-mapping>
    <filter-name>NakedObjectsStaticContentFilter</filter-name>
    <url-pattern>*.css</url-pattern>
  </filter-mapping>
  <filter-mapping>
    <filter-name>NakedObjectsStaticContentFilter</filter-name>
    <url-pattern>*.png</url-pattern>
  </filter-mapping>
  <filter-mapping>
    <filter-name>NakedObjectsStaticContentFilter</filter-name>
    <url-pattern>*.jpg</url-pattern>
  </filter-mapping>
  <filter-mapping>
    <filter-name>NakedObjectsStaticContentFilter</filter-name>
    <url-pattern>*.gif</url-pattern>
  </filter-mapping>
  ...
</web-app>
```

Listeners

Two listeners are required, one for *Restful Objects*/Naked Objects and one for RestEasy. These both bootstrap the respective libraries:

```
<web-app>
  ...
  <listener>
    <listener-class>org.nakedobjects.webapp.NakedObjectsWebAppBootstrapper</listener-class>
  </listener>

  <listener>
    <listener-class>org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap</listener-
class>
  </listener>
  ...
</web-app>
```

Servlets and Servlet Mappings

Finally, we have the servlets:

```
<web-app>
  ...
  <servlet>
    <servlet-name>RestEasyDispatcher</servlet-name>
    <servlet-class>org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-
class>
  </servlet>

  <servlet>
    <servlet-name>ResourceServlet</servlet-name>
```

```
    <servlet-class>org.nakedobjects.webapp.servlets.ResourceServlet</servlet-class>
  </servlet>
  ...
</web-app>
```

The main servlet is RestEasy's `HttpServletDispatcher` servlet, that handles all inbound requests. In addition though we configure a Naked Objects' `ResourceServlet` to serve up Javascript, CSS and images (the same as those decorated by the `NakedObjectsStaticContentFilter`).

The mappings for these servlets are:

```
<web-app>
  ...
  <servlet-mapping>
    <servlet-name>RestEasyDispatcher</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>

  <servlet-mapping>
    <servlet-name>ResourceServlet</servlet-name>
    <url-pattern>*.js</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>ResourceServlet</servlet-name>
    <url-pattern>*.css</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>ResourceServlet</servlet-name>
    <url-pattern>*.png</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>ResourceServlet</servlet-name>
    <url-pattern>*.jpg</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>ResourceServlet</servlet-name>
    <url-pattern>*.gif</url-pattern>
  </servlet-mapping>

</web-app>
```

5.3. Testing

There are several ways you can run up the webapp.

Using NakedObjectsWebServer

The Naked Objects framework provides a simple bootstrapper class that will run an embedded instance of Jetty against the `web.xml`. By default the reference to this bootstrapper is deliberately excluded, so to use it comment back in the reference to `org.nakedobjects.core:webserver`:

```
<dependency>
  <groupId>org.nakedobjects.core</groupId>
  <artifactId>webserver</artifactId>
</dependency>
```

To run, just run `org.nakedobjects.webserver.WebServer`. This takes the following arguments:

- --deploymentType (defaults to SERVER)

- --port (defaults to 8080)
- --address (defaults to localhost)
- --webapp resourceDirectory

For testing purposes you'll probably want EXPLORATION, which means you don't have to worry about authentication. For production you'll probably want SERVER, in which case you *do* need to decide how to handle authentication; but we'll discuss this more in Section 5.4, “Authentication”.

Using Maven Jetty plugin

Using the maven-jetty-plugin, we can use Maven to run an embedded instance of Jetty. First, add:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.mortbay.jetty</groupId>
      <artifactId>maven-jetty-plugin</artifactId>
      <configuration>
        <contextPath>/</contextPath>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Note

Due to a bug in *Restful Objects*, it is currently necessary to deploy to the root context (`/`).

Next, to run in EXPLORATION mode, update the `web.xml`:

```
<web-app>
  ...
  <context-param>
    <param-name>deploymentType</param-name>
    <param-value>EXPLORATION</param-value>
  </context-param>
  ...
</web-app>
```

As we've already mentioned, for testing you'll probably want EXPLORATION, while for production you'll probably want SERVER. See Section 5.4, “Authentication” for more on this.

Note

In case you were wondering, `NakedObjectsWebServer` (the section called “Using NakedObjectsWebServer”) does not currently honour deploymentType setting.

You can then run Jetty from Maven using:

```
mvn jetty:run
```

Deploy to an External Servlet Container

Finally, you can copy the WAR file and deploy it to an external servlet container.

Note

Again, because of a bug in *Restful Objects* you will need to map it to the root context for this to work.

5.4. Authentication

The RESTful architecture doesn't have too much to say about authentication, and (at the time of writing) there are no supplementary standards to define how authentication should be implemented. And because REST runs over HTTP we don't get any particular support from the network stack either. In fact, because HTTP is stateless we don't even get the notion of a session or a connection.

Restful Objects therefore uses Naked Objects' authentication mechanism, with authentication performed for each RESTful call. What this means depends on the deploymentType:

- if running in SERVER_EXPLORATION mode, then authentication is in effect switched off; no credentials are supplied, and Naked Objects will use the first exploration user defined in nakedobjects.properties, or a fallback "exploration" user otherwise

  For example, if running in SERVER_EXPLORATION mode, then you can specify the user using:

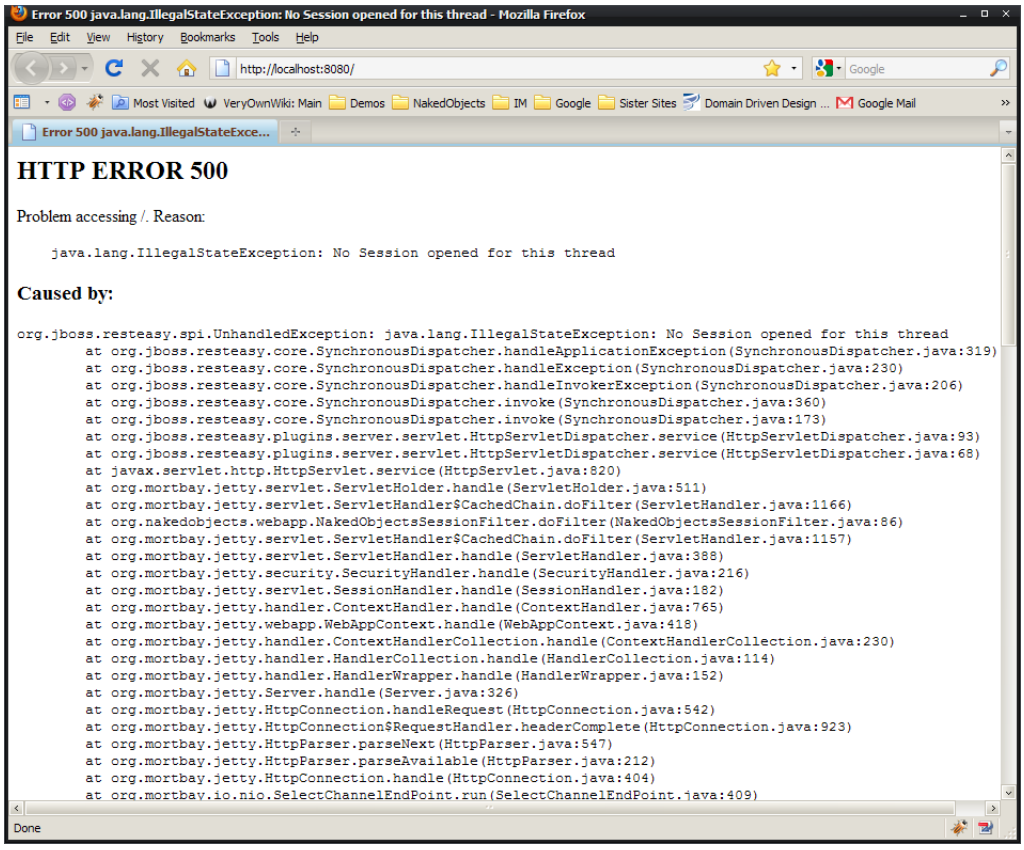
  ```
  nakedobjects.exploration.users=sven:role1, dick:role2, bob:role1|role2
  ```

- otherwise (if running SERVER_PROTOTYPE or SERVER), then authentication credentials are needed.

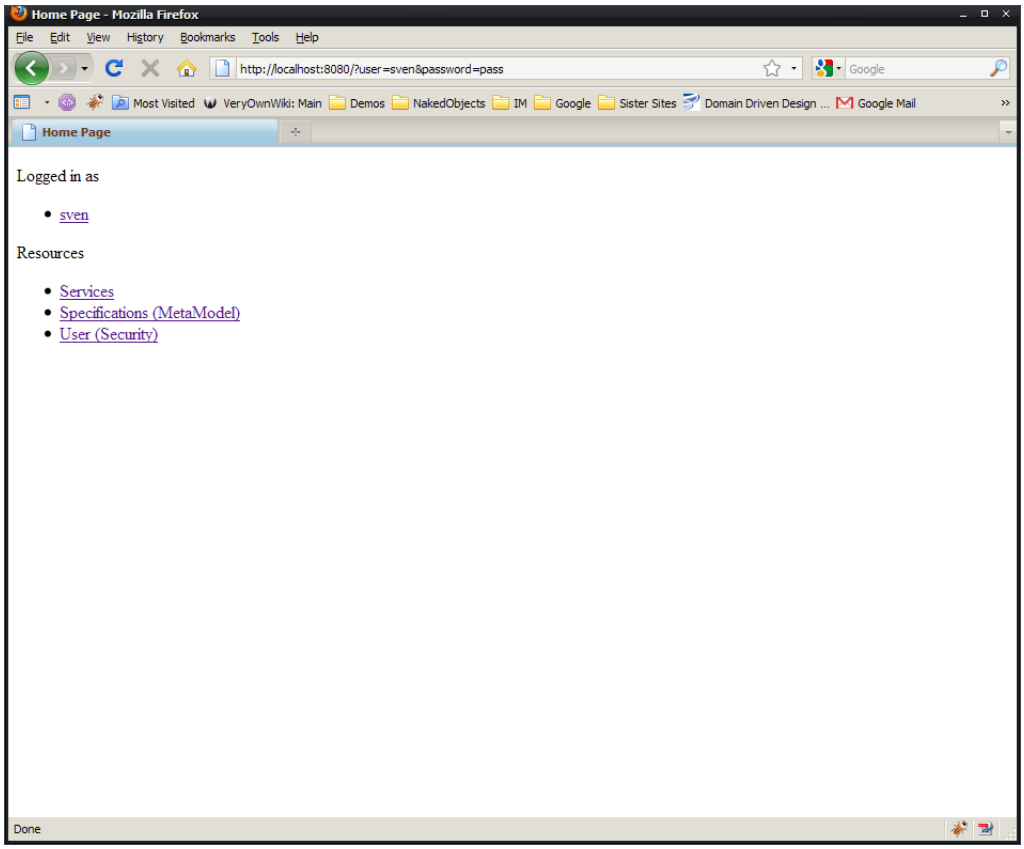
Note

You can if you want run in EXPLORATION (or PROTOTYPE) mode rather than SERVER_EXPLORATION (or SERVER_PROTOTYPE) mode. The difference is that former only support single-users, while SERVER_* supports multiple concurrent users.

Calling Restful Objects in SERVER mode with no credentials will result in an exception:



On the other hand, if user and password parameters are supplied, then we can login:



This behaviour is pluggable however, at two levels:

- `NakedObjectsSessionFilter` uses the `authenticationManagerLookupStrategy` property to specify a strategy for both finding credentials and for validating them against the Naked Objects authentication manager.

  The implementation provided by *Restful Objects* - `AuthenticationSessionLookupStrategyExtended` in the package `org.starobjects.restful.viewer.authentication` - is responsible for looking up credentials using the username and password parameters. Once validated, it also binds the results to the `HttpSession` so that future interactions do not credentials.
- If you lookup strategy implementation delegates to the Naked Objects authentication manager (recommended), you might also want to change the Naked Objects' `AuthenticationManager` implementation itself. This is done using the `nakedobjects.authentication` key in `nakedobjects.properties`:

  ```
  nakedobjects.authentication=com.mycompany.nakedobjects.authentication.MyAuthenticationManagerInstaller
  ```

In the future there will doubtless be standardized (WS-* style) approaches for RESTful authenticatication, but the above should provide enough flexibility in the meantime.