Using OAuth with RESTful Web Services

Martin Matula
Senior Software Development Manager, Project Jersey

Pavel Bucek
Senior Software Developer, Project Jersey

ORACLE

Table of Contents

Using OAuth with RESTful Web Services
Hands-on Lab: 24761

Introduction

OAuth is gaining a lot of popularity these days as a standard protocol for secure API authorization. We have seen several well-known service providers switching to OAuth from their own proprietary authorization schemes or from relying purely on HTTP Basic or HTTP Digest authentication, requiring end-user credentials to grant access.

In the Java world, JAX-RS (JSR 311) provides a standard API for building RESTful web services. Jersey, the production ready reference implementation of JAX-RS adds the client-side API that provides an easy way of interacting with web-based services (even in cases when those services are not fully adhering to the REST principles).

In this lab you are going to learn, how you can use Jersey and its extensions to build service providers and consumers secured using the OAuth protocol.

Prerequisites

This hands-on lab assumes you have a basic experience with the following technologies:

- Java SE Platform
- REST, the architectural style
- JAX-RS: Java API for RESTful Web Services

Notation Used In This Documentation

<lab_root>	The directory into which the lab contents were placed, i.e. where this document lives

Lab Exercises

This lab provides the following exercises:

1. “Hello World” Using JAX-RS and Jersey – this is an optional exercise for those who are not familiar with JAX-RS and Jersey.
2. Twitter Client Using Jersey and OAuth – shows how to build a simple Twitter client. We provide detailed steps on how to build a simple twitter client web application. The solutions directory of the lab zip file also contains an equivalent command-line twitter client application – we advise to study that as well to see the differences in the OAuth flow between the web and desktop applications.
3. Enabling OAuth for an Existing Service – guides through adding OAuth support to an existing web application.

Additional Resources

- http://oauth.net/ - OAuth web site
- http://jersey.java.net/ – Project Jersey web site
- http://wikis.sun.com/display/Jersey – Jersey wiki
- http://dev.twitter.com/ - Twitter API pages
- users@jersey.java.net – mailing list for any technical questions or feedback on JAX-RS / Jersey

Lab Environment Setup

Please refer to the Appendix at the end of this document for the detailed information on the hardware and software requirements as well as instructions on how to set up the lab environment.

Please feel free to seek assistance from the instructors or Oracle Demo staff at any point during the lab.

Exercise 1: “Hello World” Using JAX-RS and Jersey – *optional*

This exercise provides a quick introduction to the basics of the JAX-RS/Jersey programming model. For more details, please refer to the Getting Started section of the Jersey User Guide. If you are familiar with the REST architecture principles and Jersey framework, you may skip this exercise.

In this lesson you are going to build a very basic "Hello world" type of RESTful web service. It will provide an introduction to how you can quickly get started building RESTful services using Jersey framework. You will see the basic concepts of the JAX-RS/Jersey programming model.

Background Information

What are RESTful Web Services?

RESTful web services are services that are built to work best on the web.

Representational State Transfer (REST) is an architectural style that specifies constraints (such as uniform interface) that, if applied to a web service, induce desirable properties (e.g. performance, scalability, and modifiability) that enable services to work best on the Web. In the REST architectural style, data and functionality are considered resources, and these resources are accessed using Uniform Resource Identifiers (URIs), typically links on the web. The resources are acted upon by using a set of simple, well-defined operations. The REST architectural style constrains architecture to the client-server model, and is designed to use a stateless communication protocol, typically HTTP. In the REST architecture style, clients and servers exchange representations of resources using a standardized interface and protocol. These principles encourage RESTful applications to be simple, lightweight, scalable and provide high performance.

RESTful web services typically map the four main HTTP methods to the operations they perform: create, retrieve, update, and delete. The following table shows the mapping:

HTTP Method	Operations Performed
GET	Get a resource
POST	Create a resource or other operations as it has no defined semantics
PUT	Create or update a resource
DELETE	Delete a resource

Introducing Project Jersey

Project Jersey is an open source, production quality reference implementation for a standard defined in JSR-311: JAX-RS: The Java API for RESTful Web Services. It implements support for the annotations defined in JSR-311, making it easy for developers to build RESTful web services with Java and the Java VM. Jersey also adds additional features not specified by the JSR.

The goals and philosophy of Project Jersey are to:

- Make it simple and straightforward to build RESTful web services and clients using Java and possibly other Java VM based languages.
- Track the JAX-RS API and deliver production quality reference implementation.
- Provide value-add features on top of the JAX-RS specification.
- Expose APIs/SPIs to extend Jersey.

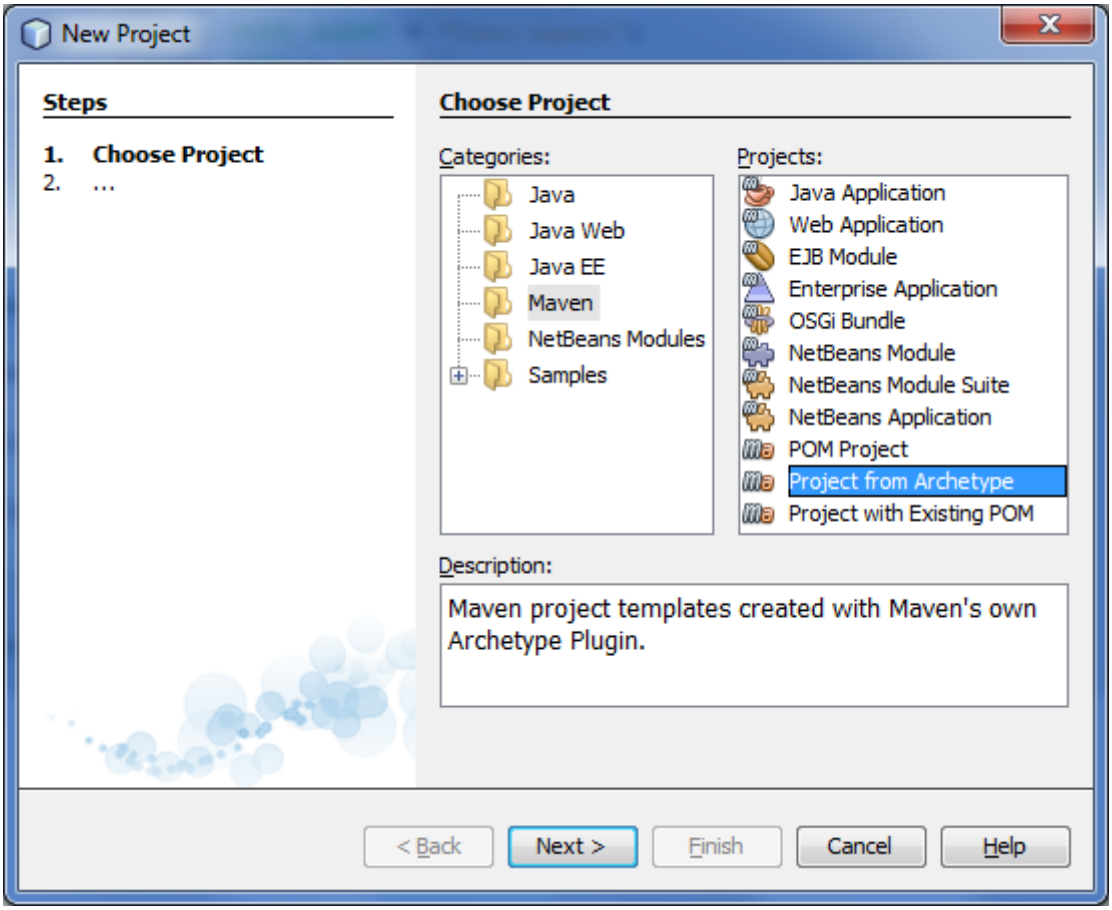
In this exercise you're going to see what it takes to build a new RESTful web service from scratch using Jersey.

NOTE: To be able to provide clear steps, we are using NetBeans to develop this lab. Anyway, all that we are going to show can easily be achieved without NetBeans, using a simple text editor or some other IDE. To demonstrate this, for operations other than creating and editing java files we provide simple instructions on how to achieve the same results from the command line.

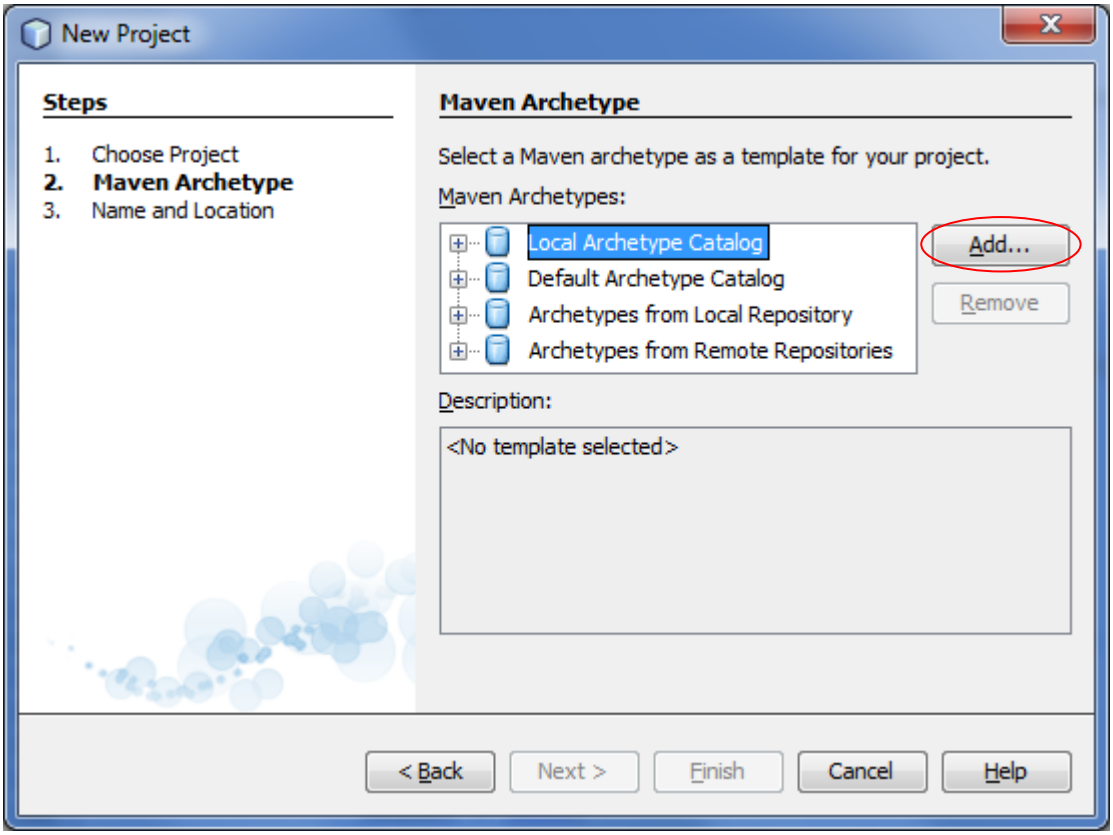
Steps to Follow

Step 1: Creating a New Project

1. If NetBeans is not already running, start it.
2. Once NetBeans has started, select File -> New Project.
3. In the New Project wizard, select Maven -> Project from Archetype and click Next.

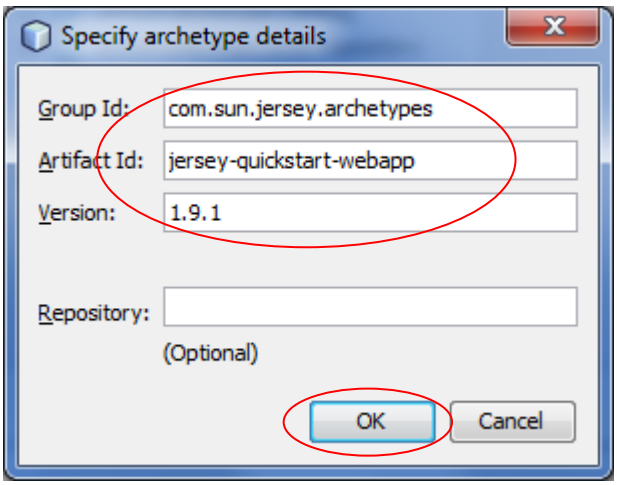


4. Click the Add button.

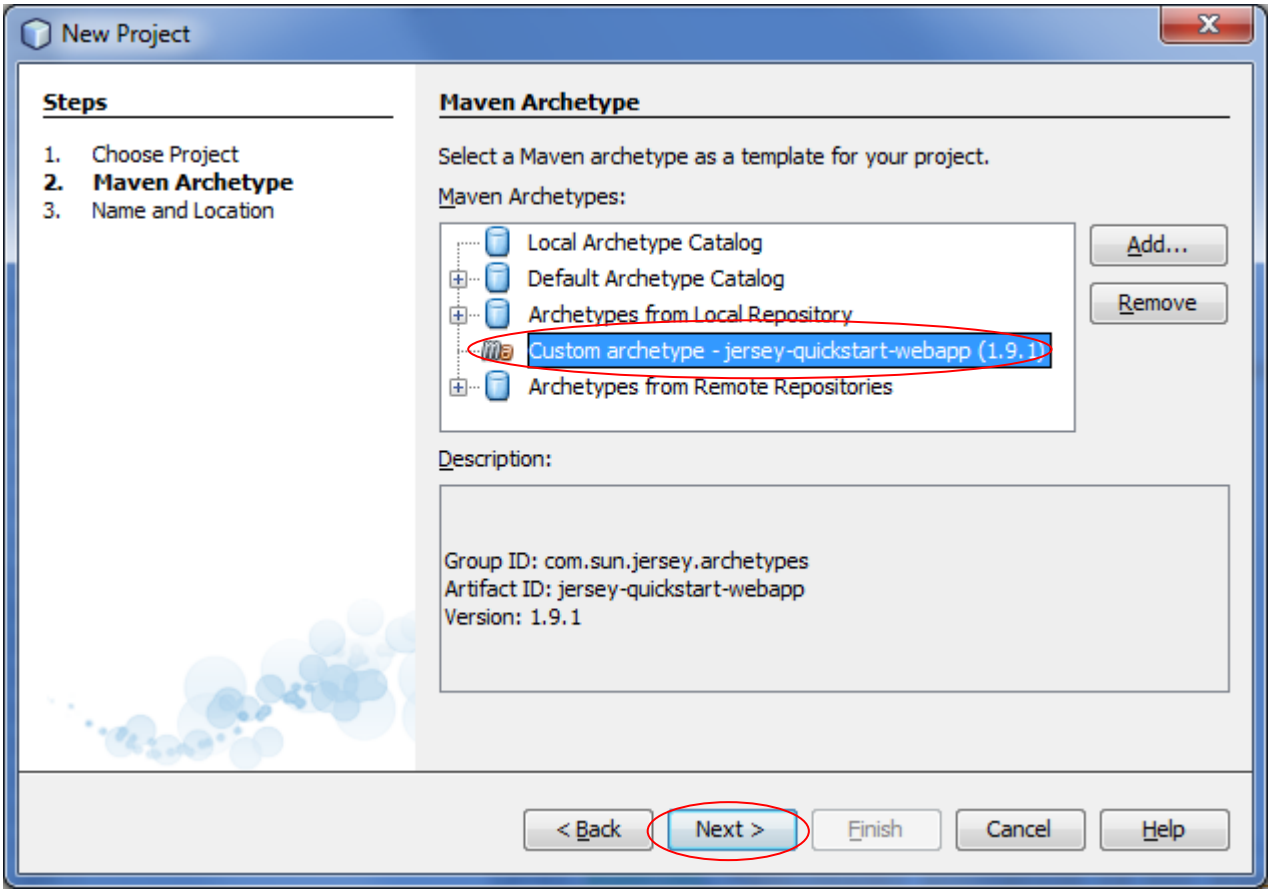


5. Dialog for the entering the archetype details will open. Fill it out as follows and click OK:

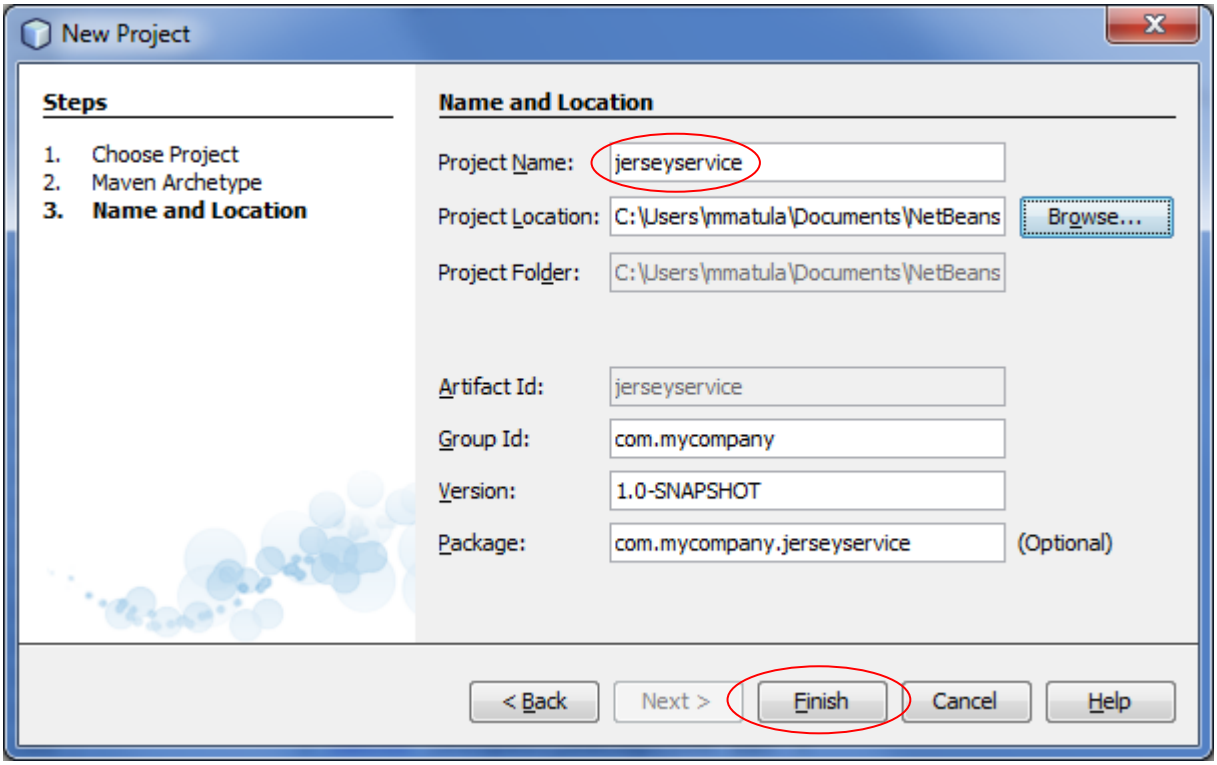
   Group Id: com.sun.jersey.archetypes
   Artifact Id: jersey-quickstart-webapp
   Version: 1.9.1



6. Back in the New Project dialog select this newly added artifact and click Next.



7. In the next screen of the wizard, name the project as “jerseyservice” and click the Finish button.



NOTE: If this is the first time you are creating a new project based on Jersey maven archetype, it may take more than a minute to create the new project after hitting the Finish button. During this time maven automatically downloads all the necessary dependencies.

Without the NetBeans IDE

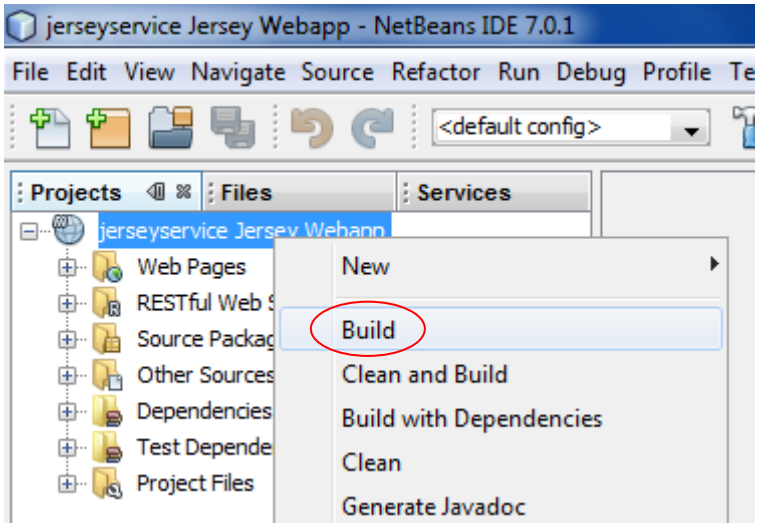
All that we’ve done so far can easily be done without NetBeans. The steps are equivalent to executing the following on the command line in the directory where you want to create the project:

```
prompt> mvn archetype:generate -DinteractiveMode=false
    -DarchetypeArtifactId=jersey-quickstart-webapp -DarchetypeVersion=1.9
    -DarchetypeGroupId=com.sun.jersey.archetypes -DgroupId=com.mycompany
    -DartifactId=jerseyservice -Dpackage=com.mycompany.jerseyservice
```

Step 2: Building the Project

Now that the new project is created, let's build it.

1. In the NetBeans right-click on the jerseyservice project – i.e. the new project we've just created – in the Projects tab and choose Build in the pop-up menu.



Since this is the first time we are building a project depending on Jersey libraries, the build may take several minutes. So, in the meantime you may go ahead to the step 3 and explore the project structure.

Without the NetBeans IDE

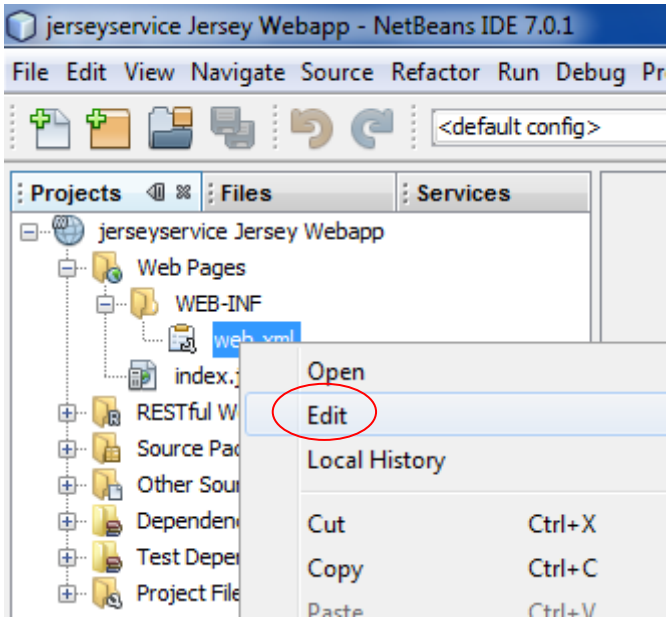
To build the project without NetBeans you can execute the following in the project root directory:

```
prompt> mvn install
```

Step 3: Exploring the Project Structure

Let's look more closely at the content of our new project.

1. Expands the Web Pages node and open index.jsp by double-clicking on it. It is a very simple page serving as a starting point for the web application the new project represents. It contains a link to a sample RESTful web service - we call it a resource - that got generated as part of this project, and also a link to the Project Jersey web site.
2. Under Web Pages -> WEB-INF open the web.xml file for editing by right-clicking on it and selecting Edit from the pop-up menu. This file is a standard web application deployment descriptor.



TROUBLESHOOTING: In case the web.xml file opened in design mode (this can happen if you double-click it instead of right-clicking and choosing Edit), switch to XML editing mode by clicking XML in the editor window.

3. Let's look at the interesting portion of the web.xml file:

```
 5 <servlet>
 6     <servlet-name>Jersey Web Application</servlet-name>
 7     <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
 8     <init-param>
 9         <param-name>com.sun.jersey.config.property.packages</param-name>
10         <param-value>com.mycompany.jerseyservice</param-value>
11     </init-param>
12     <load-on-startup>1</load-on-startup>
13 </servlet>
14 <servlet-mapping>
15     <servlet-name>Jersey Web Application</servlet-name>
16     <url-pattern>/webresources/*</url-pattern>
17 </servlet-mapping>
```

Lines 4 and 5 configure `com.sun.jersey.spi.container.servlet.ServletContainer` as the servlet class. This class is an internal class from the Jersey framework implemented to handle the HTTP requests coming from the clients and route the requests to the implementations of our resources. It knows where to look for the resource classes based on the value of the `com.sun.jersey.config.property.packages` parameter. This parameter is configured on lines 7 and 8 of the deployment descriptor to `com.mycompany.jerseyservice`, so Jersey will automatically look for the resource classes in that package and all of its nested packages.

Lines 12 to 15 then define a mapping of the servlet to a URL pattern. It tells the servlet container, requests to which URLs should be delegated to the Jersey servlet. In this case it is going to be all URLs starting with `/webresources/` right after the application context URL prefix.

NOTE: When using JavaEE6 (servlet 3.0), web.xml file is optional. Please see Jersey User Guilde to get more information on how you can avoid using web.xml file when using Jersey with servlet 3.0/JavaEE6.

4. Expand Source Packages->com.mycompany.jerseyservice and open MyResource.java by double-clicking on it. This is a simple example of a JAX-RS/Jersey resource class. It is a very simple class, just having a single method returning a string. What turns it into a resource are the annotations.

```
10 @Path("/myresource")
11 public class MyResource {
12
13     /** Method processing HTTP GET requests, producing "text/plain" MIME media
14      * type.
15      * @return String that will be send back as a response of type "text/plain".
16      */
17     @GET
18     @Produces("text/plain")
19     public String getIt() {
20         return "Hi there!";
21     }
22 }
```

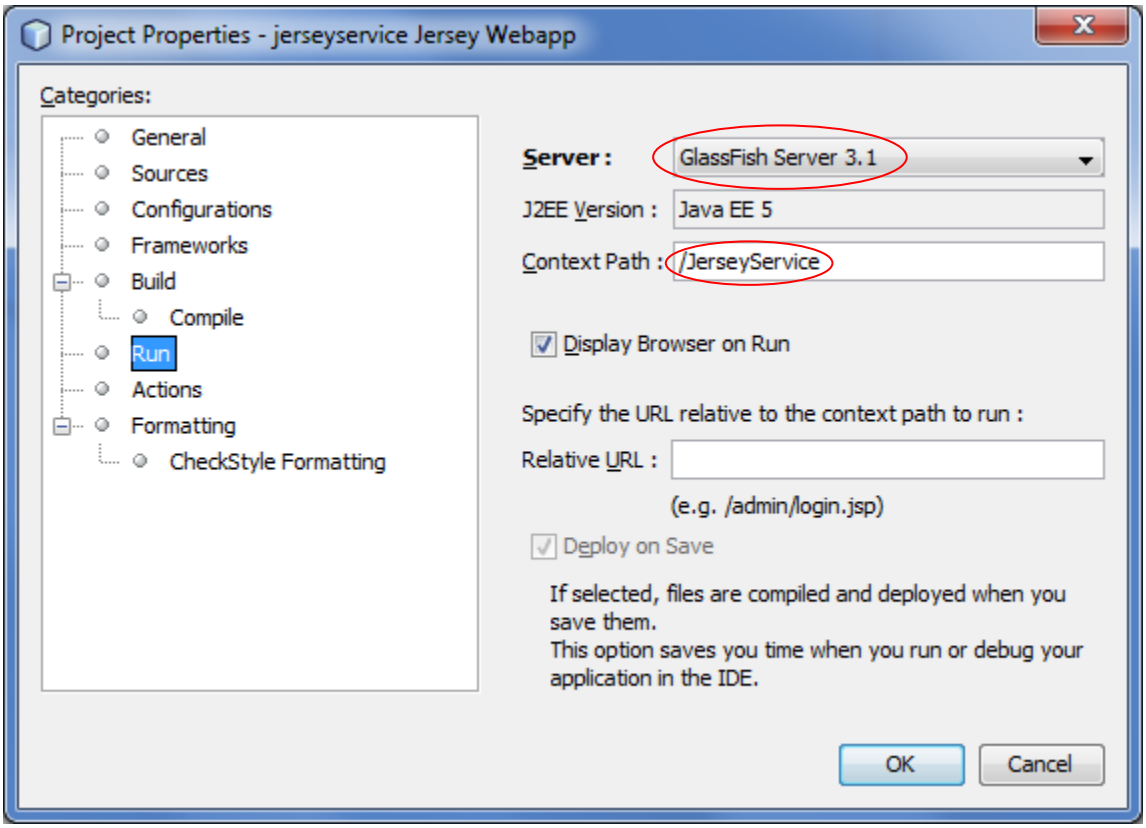
The `@Path` annotation at line 10 indicates this class is a resource available at path `/myresource` relative to the Jersey Web Application servlet URL. The method of this class is annotated by two annotations (lines 17 and 18). The first one - `@GET` - indicates this method should be invoked to handle HTTP GET requests, the second one - `@Produces` - declares the MIME type of the data the method produces (`text/plain` in this case).

5. Under Project Files you can see pom.xml. That is the file containing the maven project description. It contains references to all the dependencies based on which maven knows what jars need to be downloaded to build or test the project. More on the pom.xml file format can be found in maven documentation available from maven.apache.org.
6. The file settings.xml is also maven-specific and is out of scope of this lab.

Step 4: Running the Project

Now that we know how the project looks like and why, let's try to run it. To run the project, we first need to configure it to use GlassFish 3.1.1 as the deployment server:

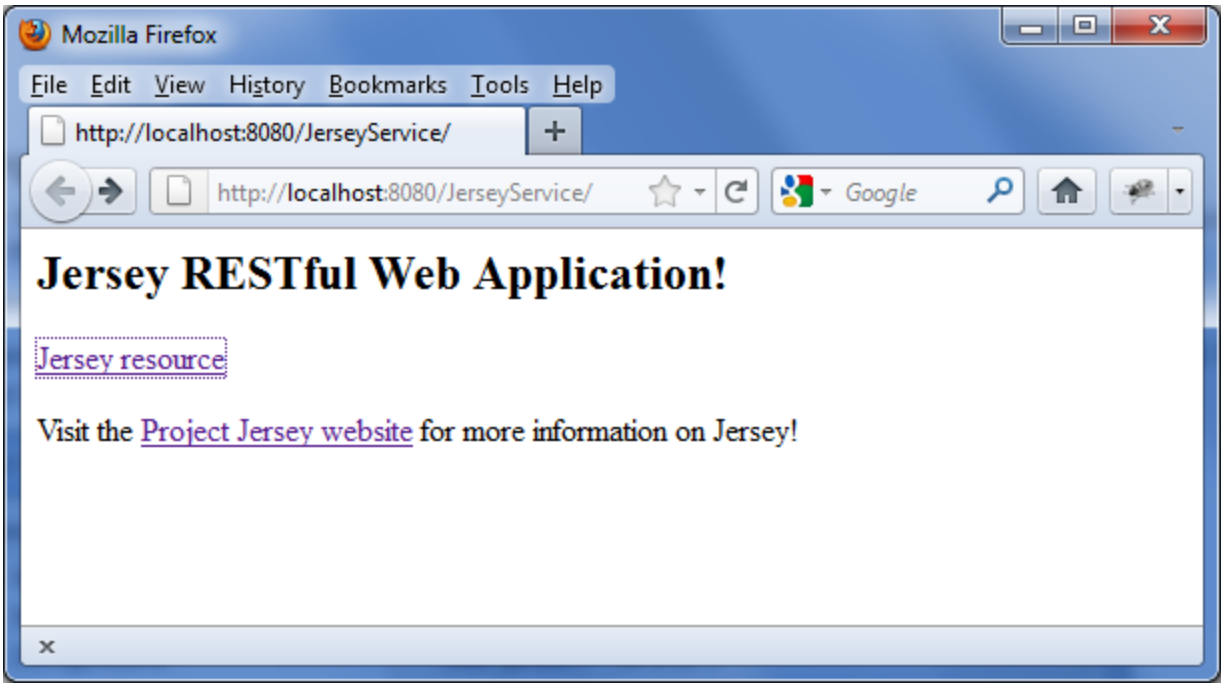
1. In NetBeans, right-click on the project and choose Properties (at the end of the pop-up menu).
2. In the Project Properties dialog select the Run category and set the Server field to GlassFish Server 3.1. Also change the Context Path to `/JerseyService`.



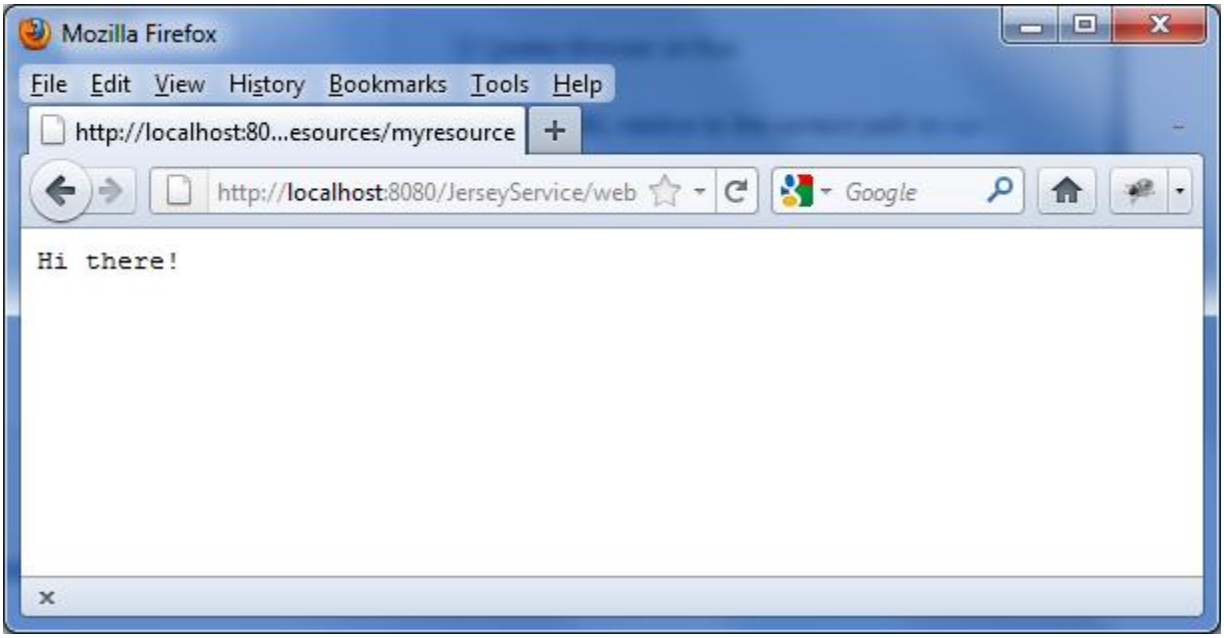
To run the project:

3. Right-click on the project and choose Run from the pop-up menu. NetBeans will create a war file containing the compiled project, start GlassFish and use it for deploying the war file.

A web browser should be started automatically, once the whole process is done, pointing to http://localhost:8080/JerseyService/. This displays the index.jsp from our project in the browser.



4. "Jersey resource" link points to our resource (MyResource). If we click on it, the browser will automatically send an HTTP GET request to a given URL which will be handled by our resource class - i.e. `getIt()` method will be called which returns a plain text resonse "Hi there!". So, let's click on the link to see if it really works.



Without the NetBeans IDE

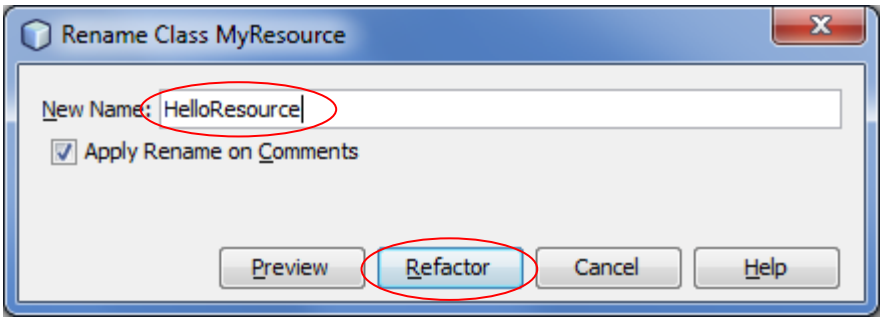
To run the project without NetBeans you can execute the following in the project root directory:

```
prompt> mvn install
prompt> asadmin start-domain
prompt> asadmin deploy target/jerseyservice.war
```

Step 5: Making Modifications

Let's see how we can customize the project a little bit:

1. Rename MyResource.java to HelloResource.java by right-clicking on it in the Projects view and choosing Refactor->Rename from the pop-up menu. Type the new name into the dialog and click Refactor.



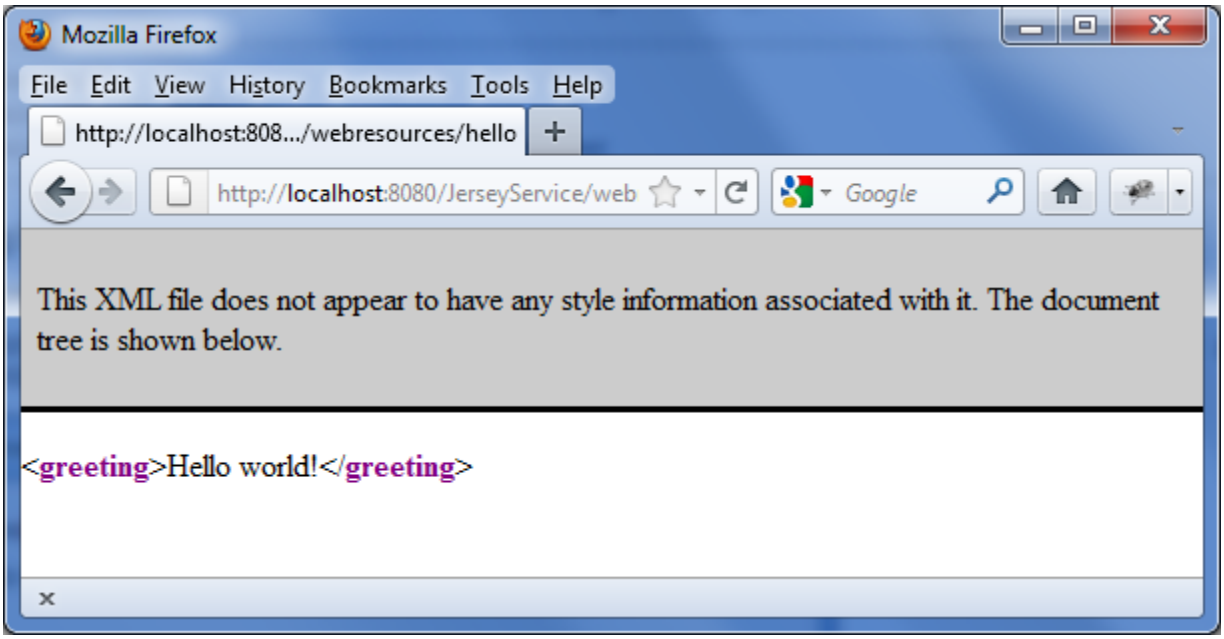
2. Open HelloResource.java (if not open yet) and change the URI template (in the `@Path` annotation) for the resource from `/myresource` to `/hello`. Update the comment as well.
3. In the @Produces annotation attached to the getIt() method change the produced MIME type to text/xml and let the method return "<greeting>Hello world!</greeting>". Update the comments as well. Here is how the resulting class should look like:

```
/** Example resource class hosted at the URI path "/hello"
 */
@Path("/hello")
public class HelloResource {

    /** Method processing HTTP GET requests, producing "text/xml" MIME media
     * type.
     * @return String that will be send back as a response of type "text/xml".
     */
    @GET
    @Produces("text/xml")
    public String getIt() {
        return "<greeting>Hello world!</greeting>";
    }
}
```

4. Since we changed the URI template, we need to update the link to our resource in index.jsp. Open that file and replace line 4 with:

```
<p><a href="webresources/hello">Hello resource</a>
```

5. Run the application again - NetBeans will build and re-deploy it. We can see it now uses the new URI and the browser recognizes the MIME type of the returned value as XML.



Summary

In this exercise, we explained the basic concepts of JAX-RS and Jersey. You saw what it takes to create a simple RESTful web application. So far, we have been using only the very basic functionality of the Jersey framework to build services. Our resource supported just a single HTTP method - GET, and it did not do much. In the following exercise we are going to explore what it takes to use Jersey to build a client of a RESTful API authenticated using OAuth.

Exercise 2: Twitter Client Using Jersey and OAuth

In this exercise we will get more familiar with Jersey client API as well as the OAuth support. We are going to build a simple client for a well-known service called Twitter. Our client will authenticate and read a few tweets from the user profile.

If you don't have a twitter account, don't worry – we have a simple twitter-like web application – MicroBlog – deployed on the lab server (and also included in the lab zip file) which you can use as an alternative. It was designed in such a way that the twitter client we will develop in this exercise will be able to talk to both the real twitter as well as our MicroBlog application.

Background Information

What is OAuth?

In a scenario like the one we are going to explore in this exercise, one application (client) needs to access data and perform actions on behalf of a user in some other application (server). We need the client to get access to the user data. In the past, the client would typically ask user to share the credentials for the server. I.e. in our scenario, the user would need to share Twitter account name and password with the client application, so that it can use those credentials to connect to the Twitter service on behalf of the user. Such approach has serious disadvantages, mainly:

- No way to limit access – the client application gains full access as if it was the actual user (resource owner) and cannot be distinguished from any other clients (or the user).
- No way to revoke access just for one particular client – the only way to ensure the client (once provided with the user credentials) cannot access the server on behalf of the user anymore, is to change the user password. Consequently, such action revokes access for any other client.

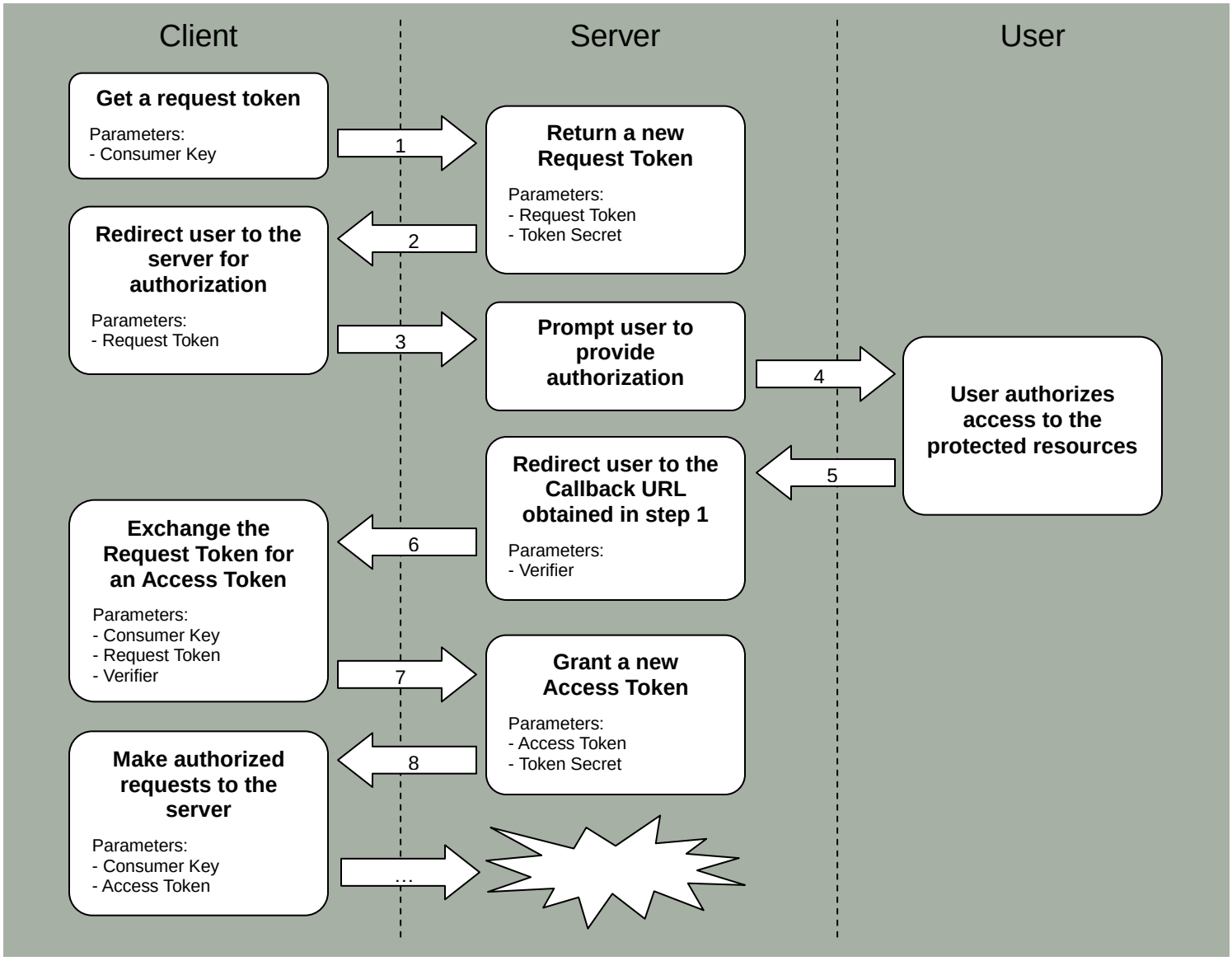
This is a client-server model, where the client application and user are indistinguishable.

OAuth makes explicit distinction between the user (resource owner) and the client. In the OAuth model, the client (which is not the resource owner, but is acting on its behalf), requests access to the resources hosted by the server. It enables the server to verify not only the resource owner authorization, but also the identity of the client making the request. Client never gets access to the resource owner's credentials. Instead it obtains an authorized token and the corresponding shared secret (once the resource owner grants access to this client), which it then uses to access the resources on behalf of the owner. Unlike the resource owner's credentials, the tokens can be issued with a restricted scope and limited lifetime, are specific to a single client and can be revoked independently.

How it works

First (before it is even deployed), the client needs to register with the server and provide the details about itself (typically the name, URL, etc.) to obtain a consumer key and a consumer secret. These are later used to identify the client making the request and are typically hardcoded in the client.

Later, when the client needs to access the resources on the server on owner's behalf, it has to attach an authorized token to its requests. The diagram on the following page illustrates the process of obtaining the authorized token. We will refer to the numbered steps in this diagram throughout this exercise to help you understand the individual parts of the process and how they fit together.



Our Application

Corresponding to the diagram above, our application is the „Client“ and Twitter service is the „Server“. User is you.

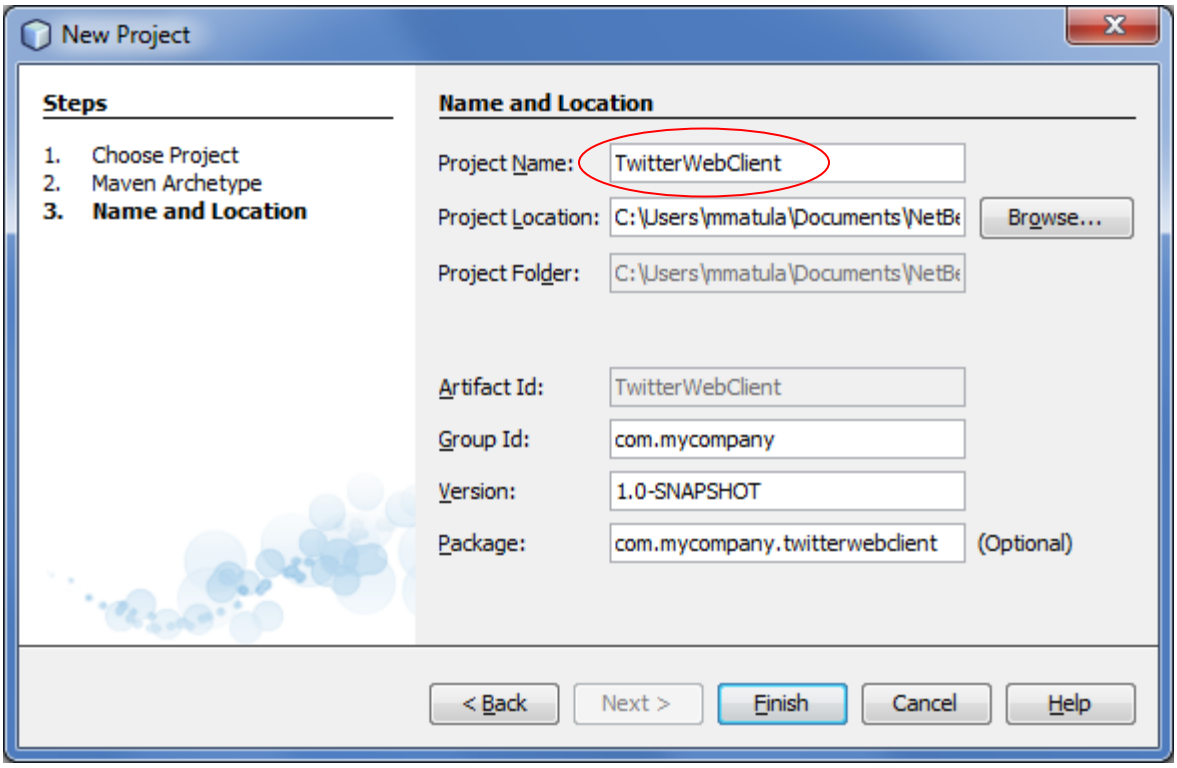
The application is going to be very simple – a web application, which will have just one link on its main page: My Feed. That link will navigate to the /feed resource of the application which will read the last few status updates from your Twitter (or MicroBlog) account and render them on the page.

When serving the /feed resource, the application will have to contact Twitter (or MicroBlog) to retrieve data on behalf of the user. For that, it will need to get authorized by the user and will have to obtain an authorized token from the server it would attach to a subsequent request to the server. This sounds complex, but as you will see, most of the steps in this process will remain transparent to you, as the Jersey OAuth Client library does a good job of hiding the OAuth complexity.

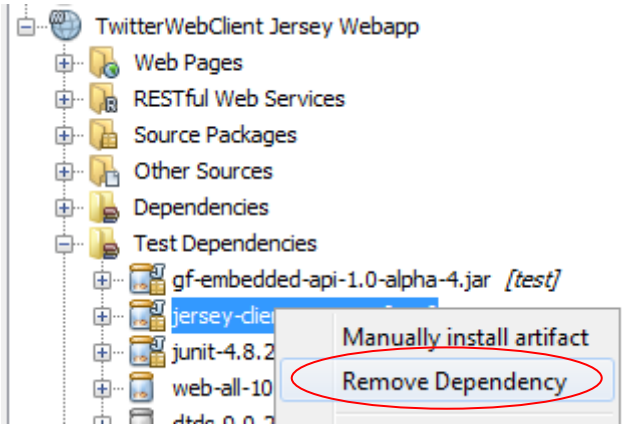
Steps To Follow

Step 1: Creating a New Project

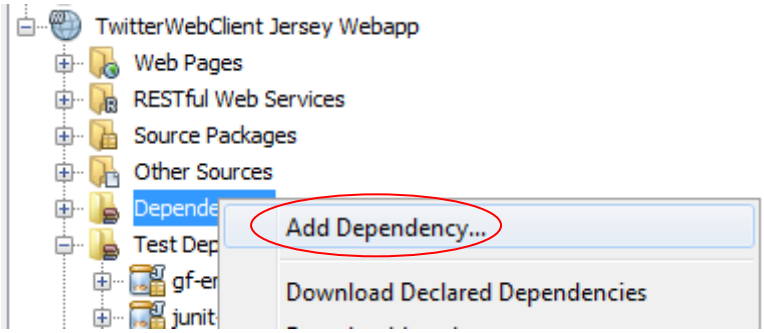
1. Like in the previous exercise, we are going to create a new project – as the start, follow the Step 1, bullet 1 to bullet 6 from the first exercise (pages 7 to 9).
2. In the next screen of the New Project wizard enter TwitterWebClient as the project name and click the Finish button.



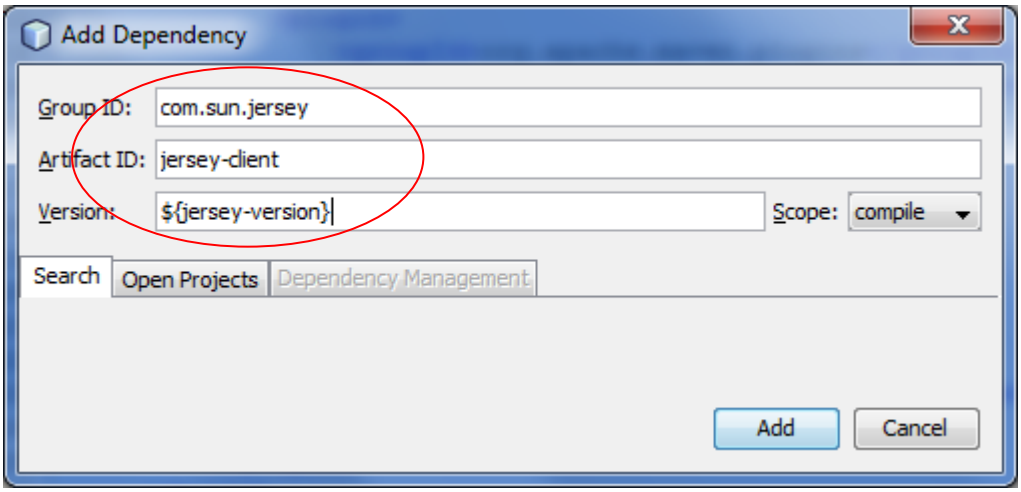
3. Once the project is created, we need to adjust the dependencies – introduce dependencies on Jersey Client API library and Jersey OAuth libraries. Jersey Client API is used in tests for the default project created from an archetype. We need to remove the dependency from the Test Dependencies and add it to the Dependencies. To remove it from the Test Dependencies, expand the Test Dependencies node in the Project Explorer, right-click on the jersey-client-1.9.jar and click on Remove Dependency in the pop-up menu, and confirm removal by clicking OK in the confirmation dialog.



4. Now we will add the dependencies to the Dependencies node. First for the jersey-client. Right-click on the Dependencies node under the project node and click Add Dependency in the pop-up menu.



5. In the dialog, enter com.sun.jersey in the GroupId field, jersey-client in the ArtifactId, ${jersey-version} in the Version field and click OK.



The Version can be set to a particular version available in the Maven repository, however here we decided to leverage ${jersey−version} variable which ensures that Maven picks the version corresponding to the other Jersey libraries our project is using (this variable was defined by the Jersey Web Application archetype we used to create the project).

6. Now we need to follow the same steps to add a dependency on oauth-client module. This module provides a client filter class, that can be used to attach the appropriate OAuth header to the out-bound client requests. So, right-click on the Libraries node again, in the dialog set the fields to the following values:

   GroupId: com.sun.jersey.contribs.jersey-oauth
   ArtifactId: oauth-client
   Version: ${jersey-version}

7. And finally we'll add a dependency on oauth-signature library which adds support for OAuth message signing.

   GroupId: com.sun.jersey.contribs.jersey-oauth
   ArtifactId: oauth-signature
   Version: ${jersey-version}

8. Rebuild the project by right-clicking on the project node and clicking on Clean and Build in the pop-up menu. As part of the build process, maven will download all the missing dependencies.

Step 2a: Requesting Consumer Key and Consumer Secret from Twitter

As mentioned in the OAuth description, every client (our Twitter client application in this case) needs a Consumer Key and the corresponding Consumer Secret. This makes it possible for the server to identify the client. We have to request this info from the server (Twitter in this case) and use these values in our Twitter client application to communicate with the server.

> NOTE: If you don't have a twitter account, or have issues completing this step with Twitter, go to Step 2b to be guided through the steps of using our simple MicroBlog service instead.

Please follow these steps to register your own application and receive a unique Consumer Key/Secret pair:

1. In a web browser, visit the following Twitter Application Registration URL: http://twitter.com/apps/new
2. Log in.
3. In the following page, fill in the registration details. Note that your application name should be unique, so use something less generic – like what we suggest below. Leave the remaining values/fields at default.

   Name: <YOUR_ TWITTER_ID>'s OAuth Jersey Client

   Description: Twitter client used for the purpose of learning how to use OAuth.

   Application WebSite: http://java.net

   Callback URL: http://127.0.0.1:8080/twitterclient/webresources/authorized

   Out of the values above, the most important is „Callback URL". That is the URL which Twitter will redirect to after successful authentication. It is the URL used in Step 6 in the flow diagram at the beginning of this exercise.

Home → My applications

Create an application

Application Details

Name: *

Matulic's OAuth Jersey Client

Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens.

Description: *

Twitter client used for the purpose of learning how to use OAuth.

Your application description, which will be shown in user-facing authorization screens.

WebSite: *

http://java.net

Your application's publicly accessible home page, where users can go to download, make use of, or find out more information about your application. This fully-qualified URL source attribution for tweets created by your application and will be shown in user-facing authorization screens.
(If you don't have a URL yet, just put a placeholder here but remember to change it later.)

Callback URL:

http://127.0.0.1:8080/twitterclient/webresources/authorized

Where should we return after successfully authenticating? For @Anywhere applications, only the domain specified in the callback will be used. OAuth 1.0a applications sho specify their `oauth_callback` URL on the request token step, regardless of the value given here. To restrict your application from using callbacks, leave this field blank.

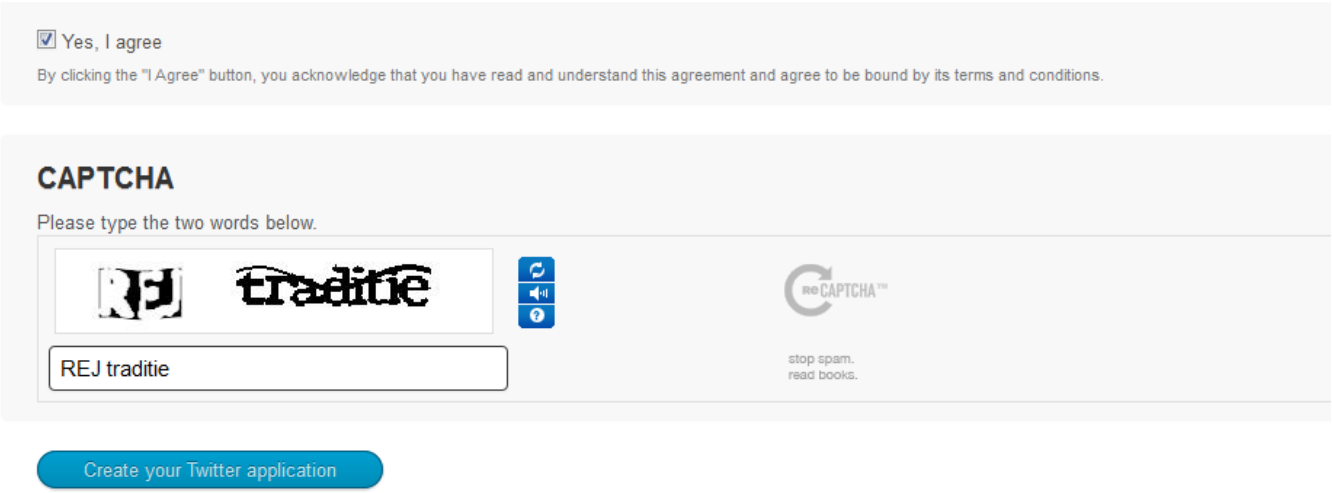
Developer Rules Of The Road

claim. In any event, you will not settle any such claim without Twitter's prior written consent.

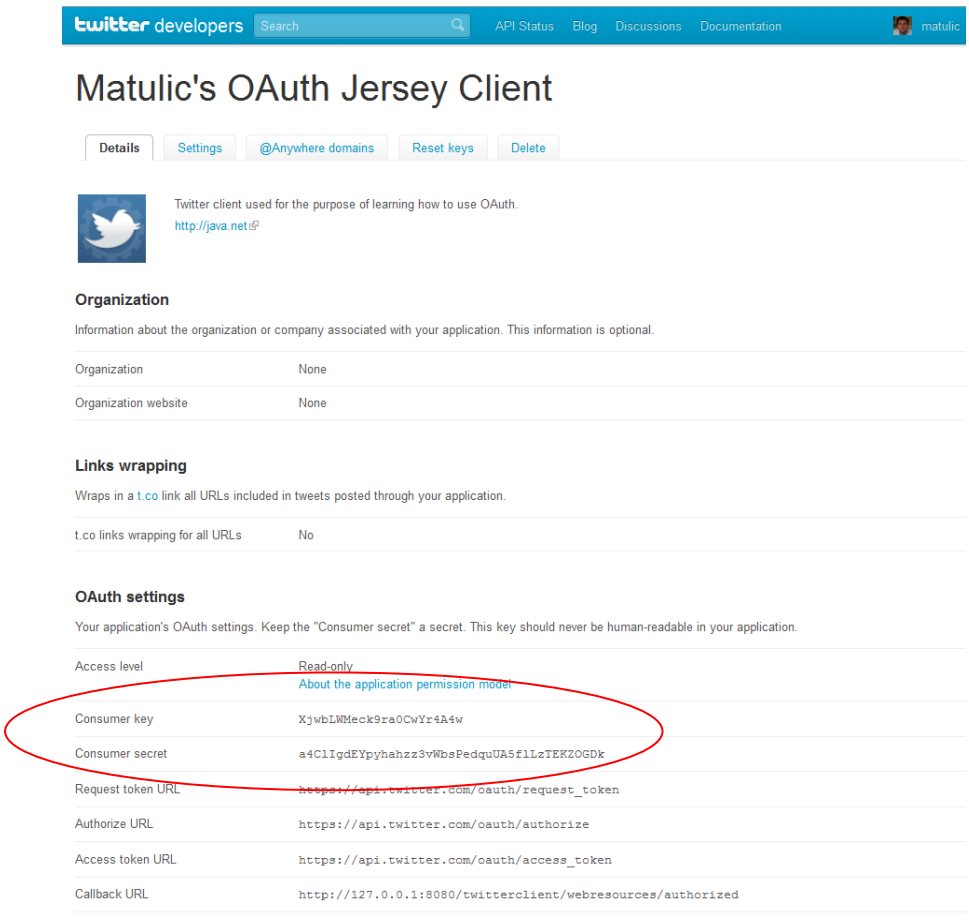
8. Miscellaneous.

These Rules constitute the entire agreement among the parties with respect to the subject matter and supersedes and merges all prior propos understandings and contemporaneous communications. Any modification to the Rules by you must be in a writing signed by both you and Twi You may not assign any of the rights or obligations granted hereunder, voluntarily or by operation of law (including without limitation in connecti a merger, acquisition, or sale of assets) except with the express written consent of Twitter, and any attempted assignment in violation of this paragraph is void. This agreement does not create or imply any partnership, agency or joint venture. This agreement will be governed by and c in accordance with the laws of the State of California, without regard to or application of conflicts of law rules or principles. All claims arising ou relating to this agreement will be brought exclusively in the federal or state courts of San Francisco County, California, USA, and you consent personal jurisdiction in those courts. No waiver by Twitter of any covenant or right under this agreement will be effective unless memorialized in

4. Check the „Yes, I agree“ field, fill in the captcha field at the bottom and click Create your Twitter application.



5. If everything went fine, you will receive your Consumer Key, Consumer Secret and a set of URLs specific to Twitter implementation of OAuth service. The page should look similar to this one:



Step 2b: Requesting Consumer Key and Consumer Secret from MicroBlog

The lab instructors will share the URL of the lab server. If you are trying this at home, you can find the MicroBlog application in <lab_root>/microblogsvc directory and run it on your machine (from NetBeans) – then it will be reachable at http://localhost:8080/microblog.

Here are the steps you should follow to get the consumer key and consumer secret:

1. On the MicroBlog main page you will be presented with a Login form. Click the Register link in the navigation bar above the login form.
2. Think of some unique name (the app does not check if the name is in use) – for simplicity you can use your full name and empty password to register for an account with MicroBlog.

> IMPORTANT: The password you enter is not encrypted, so don't use any of your real passwords in case you want to enter a non-empty one.

3. Once you register, you will be presented with the login screen again – log in using the User name and Password you registered with.
4. Click the Register App link in the navigation bar (the rightmost item).
5. Enter the desired display name of the client application (e.g. User name's OAuth Client) and click OK.
6. You will be presented with the Consumer Key and Consumer Secret.

Step 3: Designing the Main Page

The main page is index.jsp. As we mentioned earlier, all it will contain is a link to „feed“ URL – clicking the link will send a GET request to the /feed resource of our client application. The /feed resource will access your Twitter/MicroBlog account to read a few messages and return them as an HTML web page back to your browser.

When making the request to Twitter/MicroBlog, our client will have to authenticate using OAuth and get authorized by you to access your messages on the server. Behind the scenes it needs to request a so called Request Token, then redirect to Twitter/MicroBlog to get authorized by you. Once you authorize, Twitter will redirect the browser back to the callback URL you supplied during the registration of your application in case of Twitter or the URL that the application will send with the authorization request (in case of MicroBlog). For that, we need to create another resource (besides /feed), which will used as the mentioned callback. We will make it available at /authorized path. Once the Request Token is authorized, our client will exchange it for an authorized Access Token and that will be used for all the subsequent requests.

All the steps of the flow mentioned above happen behind the scenes and are handled by the Jersey OAuth client filter – all we will have to do is providing the /authorized resource.

So, to sum it up, here are the resources our application will expose:

Path (from the resource root)	**Resource Class**	**Operation**	**Description**
/authorized	AuthorizedResource	GET	Callback the server (Twitter or MicroBlog) will redirect to after the user authorized our client. Redirects back to /feed resource.
/feed	FeedResource	GET	Retrieves and returns the last few messages from Twitter/MicroBlog.

The main page – index.jsp – will be static with the single link to /feed resource. Let’s update it:

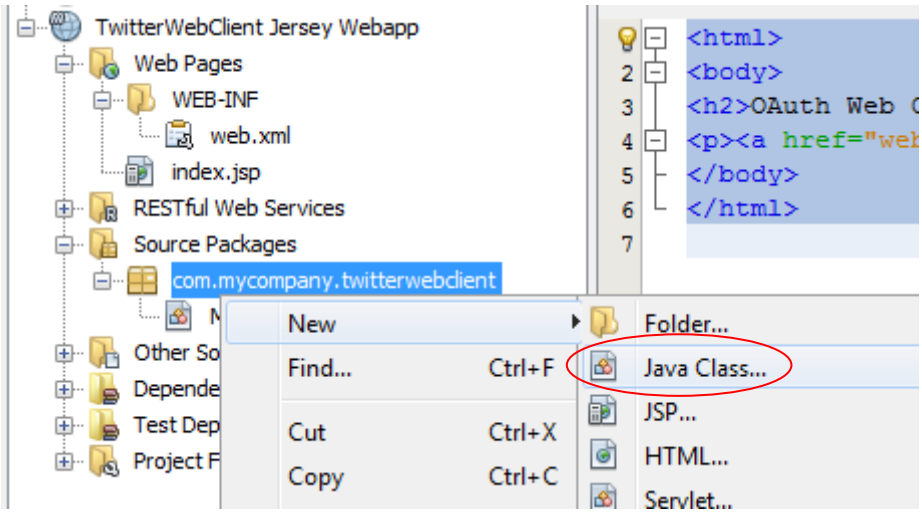
1. Expand Web Pages node in the project, and open index.jsp by double-clicking on it.
2. Enter „OAuth Web Client“ instead of Jersey RESTful web application and fill in the rest of the page to match the code below.

```
<html>
<body>
<h2>OAuth Web Client</h2>
<p><a href="webresources/feed">My Feed</a>
</body>
</html>
```

Step 4: Implementing the /feed Resource

Next, we are going to implement the /feed resource. First we will add a helper class, we will use for storing application configuration and use it as a data cache.

1. Right-click the Source Packages->com.mycompany.twitterwebclient package in the Project view, and select New->Java Class.



2. Name the class „Settings“ and click Finish.
3. Next, add the following constants to the class. The fields contain all the information we obtained from Twitter, such as Consumer Key/Secret pair (replace the values below with your Consumer Key/Secret pair that you obtained from Twitter during application registration), OAuth implementation specific URLs (REQUEST_TOKEN_URL, ACCESS_TOKEN_URL, AUTHORIZE_URL), and RESTful resource from Twitter api which we are going to use to retrieve the status list from user's account (FRIENDS_TIMELINE_URL).

```
private static final String CONSUMER_KEY = <your consumer key>;
private static final String CONSUMER_SECRET = <your consumer secret>;
```

4. A) If you are using Twitter, add also the following:

```
private static final String REQUEST_TOKEN_URL =
"https://api.twitter.com/oauth/request_token";
private static final String ACCESS_TOKEN_URL =
"https://api.twitter.com/oauth/access_token";
private static final String AUTHORIZE_URL = "https://api.twitter.com/oauth/authorize";
public static final String FRIENDS_TIMELINE_URL =
"http://api.twitter.com/1/statuses/friends_timeline.xml";
```

B) If using MicroBlog, add the following:

```
private static final String MICROBLOG_URL = "http://localhost:8080/microblogdemo/";
private static final String REQUEST_TOKEN_URL = MICROBLOG_URL + "requestToken";
private static final String ACCESS_TOKEN_URL = MICROBLOG_URL + "accessToken";
private static final String AUTHORIZE_URL = MICROBLOG_URL + "authorize";
public static final String FRIENDS_TIMELINE_URL = MICROBLOG_URL;
```

5. Besides these constants, we will use this class to keep a pointer to a single instance of Jersey client object we will be using to make requests to Twitter/MicroBlog.

NOTE: Sharing one Jersey client instance for all requests is an issue for a web app in general, as many simultaneous requests may be coming in. Also you can't share the same client for several users (as the client carries user-specific credentials for the service it connects to). We are making this simplification to stay focused on the primary topic of this lab.

Add the following code to the Settings class:

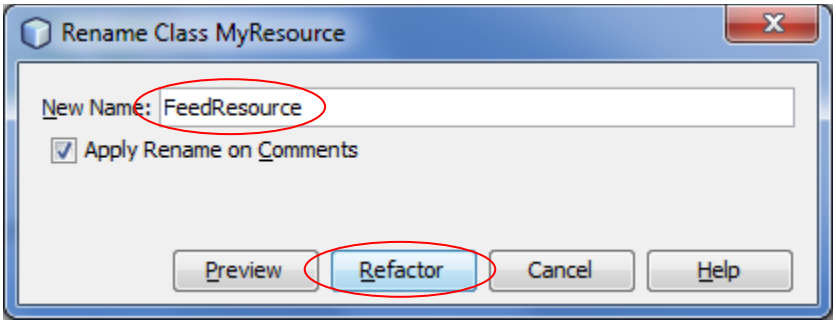
```
    private static final Client cachedClient;
    private static final OAuthParameters parameters;

    static {
        // create a new Jersey client
        cachedClient = Client.create();
        parameters = new OAuthParameters().consumerKey(CONSUMER_KEY)
                .callback("http://127.0.0.1:8080/twitterclient/webresources/authorized");

        // create a new OAuth client filter passing the needed info
        OAuthClientFilter filter = new OAuthClientFilter(
                cachedClient.getProviders(),
                parameters,
                new OAuthSecrets().consumerSecret(CONSUMER_SECRET),
                REQUEST_TOKEN_URL,
                ACCESS_TOKEN_URL,
                AUTHORIZE_URL,
                null);

        // Add filter to the client
        cachedClient.addFilter(filter);
    }

    public static Client getClient() {
        return cachedClient;
    }
```

NOTE: Don't forget to fix imports after copy-pasting the code (Ctrl+Shift+I on Windows, Cmd+Shift+I on a Mac).

As you can see, the code creates a new instance of Jersey Client and initializes it with OAuthClientFilter instance. That's a client-side request/response filter provided by the Jersey OAuth library that takes care of implementing the OAuth authorization flow. For the filter to be able to automatically request the request and access tokens as needed and to request the authorization, it needs to know the server URLs for doing so – that's what we need to pass as the constructor parameters to the filter. It also needs to know what existing OAuth parameters we want to use. At the minimum, we need to pass consumer key in OAuthParameters object and we also need to pass consumer secret in the OAuthSecrets object. If we already had an authorized access token (e.g. our client app got authorized in one of the previous session and persistent the access token for the future use), we could also load that token from a persistent store and add the token value to the OAuthParameters used to initialize the filter and the token secret into the OAuthSecrets.

Note, that besides the cachedClient, we are also keeping a reference to OAuthParameters we are passing to the filter. We will see why that's useful later.

5. Let's move on to add the main resource of our application – the /feed resource that will return the list of messages from Twitter/MicroBlog. We will refactor the existing MyResource class for that. Right-click the "MyResource" class in the Project Explorer, and select Refactor->Rename. Rename the class to "FeedResource" and click "Refactor".



6. Double-click on the FeedResource to open it in the editor. Note the @Path annotation still specifies "myresource" value. We want the resource to be available at /feed relative path (instead of /myresource). So update the @Path annotation as well as the code comments accordingly.

7. The getIt() method that is already present will serve the HTTP GET requests. We will use the client object cached in the Settings class to access Twitter or MicroBlog and read the messages which our GET method will then return as a string. Here is the code to do that – read through the comments to understand what it does:

```
@GET
@Produces("text/plain")
public String getIt() {
    // client resource representing Twitter/MicroBlog feed on the server
    WebResource r = Settings.getClient().resource(Settings.FRIENDS_TIMELINE_URL);
    // make an HTTP GET request to request a response of media type "application/xml"
    // and convert it to String
    String statuses = r.accept(MediaType.APPLICATION_XML).get(String.class);
    // return the result
    return statuses;
}
```

8. What we wrote so far would be good enough, if we had a fixed authorized access token (which we would know in advance and could hard code into our application when initializing the OAuthClientFilter). However, we don't have that. So, our request to Twitter/MicroBlog won't be authorized and would fail. OAuthClientFilter will recognize that and initiate the authorization flow behind the scenes. Instead of our GET request to retrieve the messages from the service, it will request a Request Token using the REQUEST_TOKEN_URL we supplied to the filter during the client initialization in the Settings class. Once the filter receives the Request Token from the service, we need to use that token to redirect he user to the authorization page. That cannot be done by the client filter, since the client filter does not know in what context we are using it and how to redirect a user, however it will build the right URL (containing the right query parameters as per the OAuth specification) and pass it to us by throwing UnauthorizedRequestException and including the right authorization URL in a property of that exception – so all we have to do is handle that exception and redirect the user to that URL. So, let's wrap the body of the getIt() method in a try-catch block and implement the catch section as follows:

```
try {
    <body of the getIt() method>
} catch (UnauthorizedRequestException e) {
    // throw WebApplicationException exception that is be automatically
    // mapped by Jersey to the specified response
    // (i.e. redirect to the URI obtained from e.getAuthorizationUri())
    throw new WebApplicationException(Response.seeOther(e.getAuthorizationUri())
        .build());
}
```

Step 5: Implementing the /authorized Resource

So, now we are handling the authorization failure by redirecting back to the service to ask user to authorize our request. The users will see the Twitter/MicroBlog authorization page where they will authenticate with their Twitter/MicroBlog credentials and authorize the request. Once the request is authorized, the service will redirect the user back to our application, i.e. to the callback URI we provided in OAuth parameters when initializing the Jersey client in the Settings class, and in case of Twitter also during our client registration:

http://127.0.0.1:8080/twitterclient/webresources/authorized

> NOTE: 127.0.0.1 is the same thing as "localhost". We are using the numeric form, since Twitter is not happy about callback URIs containing "localhost".

This URI corresponds to /authorized resource in our application. We haven't implemented such resource yet. So, let's do so:

1. Right click the com.mycompany.twitterwebclient package in the Project explorer, and select New->Java Class. Name the class AuthorizedResource.
2. To turn the class into a Jersey resource that will serve requests at /authorized path, add the @Path annotation with the value of "/authorized".

3. Based on the OAuth specification, the service will send us the request token this authorization corresponds to (for simplicity we are going to ignore that and assume we won't be processing several simultaneous authorizations) and a so called verifier, which we will have to add to the OAuth parameters – the OAuthClientFilter is then going to use the verifier to request the authorized access token behind the scenes upon the next request we will be making to Twitter/MicroBlog. The verifier will be passed to our client in form of a query parameter in the request URI. With Jersey/JAX-RS we can simply inject it's value into a method parameter value. So, let's add the following method to the AuthorizedResource class:

```
    @GET
    public Response get(@QueryParam(OAuthParameters.VERIFIER) String verifier) {

    }
```

4. All we have to do in this method is take the verifier, add it to the OAuth parameters used by the OAuthClientFilter and redirect back to the /feed resource to make another attempt at retrieving the messages from the service. As you know, we have a reference to the OAuth parameters used by the filter as a static field in the Settings class. But the field is private and we can't access it from AuthorizationResource. But we don't need access to the parameters, all we need is to add the verifier. So let's add a convenience method to the Settings class that will do that for us. Switch to the Settings class in the NetBeans editor and add the following method:

```
    public static void setVerifier(String verifier) {
        parameters.verifier(verifier);
    }
```

5. Finally, switch back to the AuthorizedResource class and implement the get() method so that it will call Settings.setVerifier() to set the verifier code and redirect back to the /feed resource. Here is how the whole class should look like:

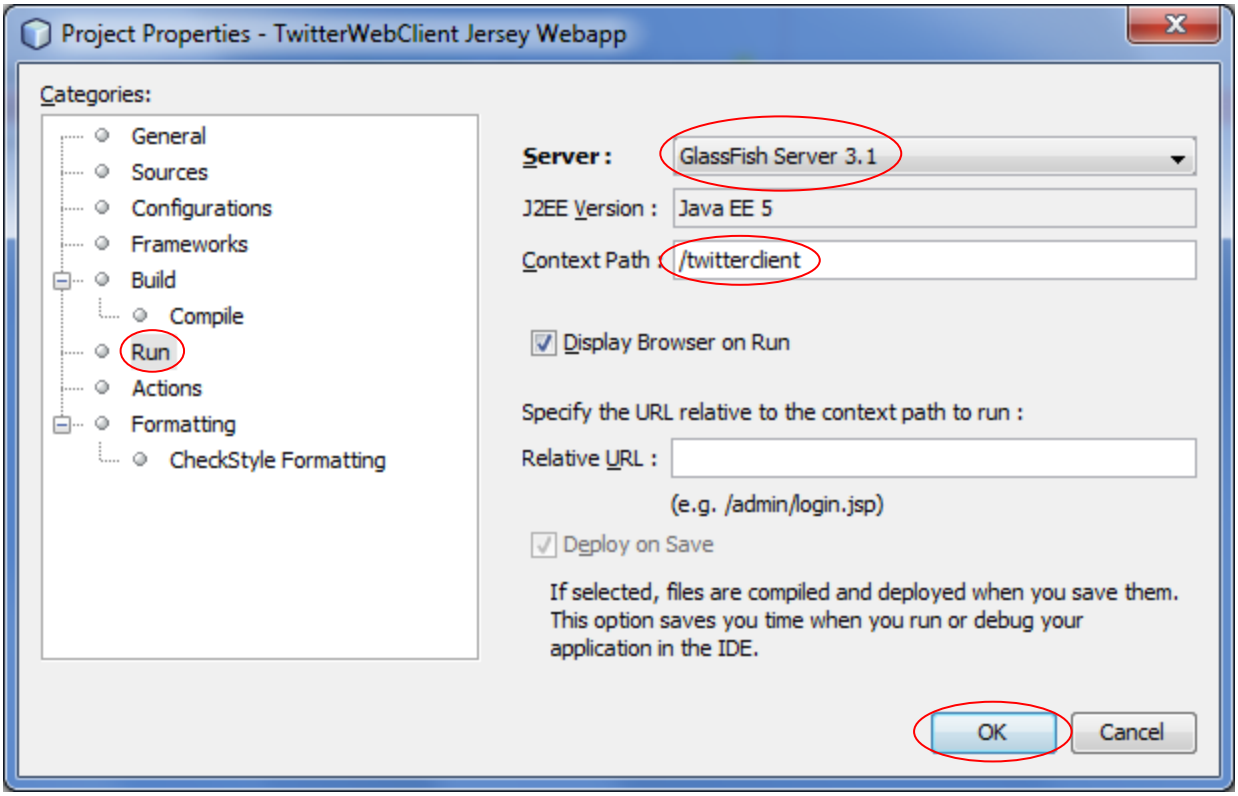
```
@Path("/authorized")
public class AuthorizedResource {
    @GET
    public Response get(@QueryParam(OAuthParameters.VERIFIER) String verifier) {
        Settings.setVerifier(verifier);
        return Response.seeOther(UriBuilder.fromPath("/feed").build()).build();
    }
}
```

With that, we finished the implementation of /authorized resource.

Step 6: Running the Application

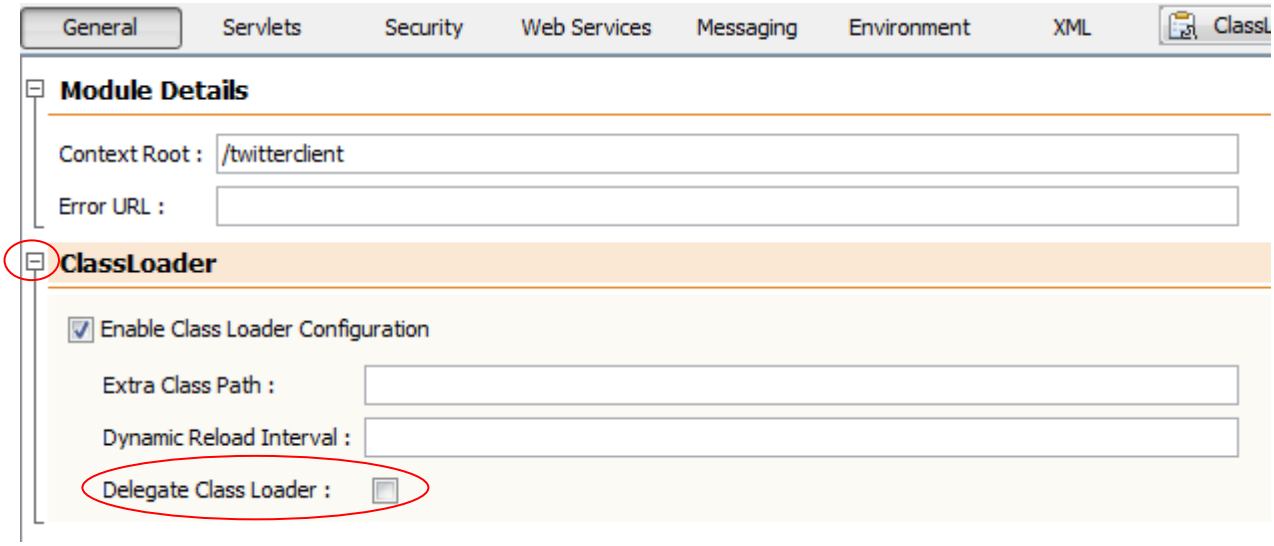
The code is ready. Now we will make a few tweaks in the configuration and run the application.

1. Right-click on the TwitterWebClient project in the project explorer in NetBeans and click on Properties in the pop-up menu. The Project Properties dialog will appear.
2. Click on Run in the left panel of the dialog and in the right panel set GlassFish 3.1 as the Server and /twitterclient as the Context Path. Then click OK. (see the screenshot on the next page)



3. When you expand Web Pages->WEB-INF folder of the TwitterWebClient project, you should now see web.xml and glassfish-web.xml files. Double-click the glassfish-web.xml file. That's the GlassFish specific deployment descriptor. A visual editor of the descriptor will appear in the editor pane.

4. Expand the ClassLoader section in the editor and uncheck the Delegate Class Loader option. This will ensure our application will be using the Jersey libraries bundled in the application rather than the ones included in GlassFish.



5. Run the application by right-clicking on the project and clicking Run in the pop-up menu. NetBeans will save all files, build the application, deploy it on GlassFish and open the main page in Firefox.
6. Click the My Feed link on the page, which will invoke the /feed resource in our application. The OAuthClientFilter will initiate the authorization flow and you will get redirected to the authorization page of the service. Log in if you are not logged in. You will be presented with a page asking you to authorize the client application. In case of Twitter it will look as follows:

7. When you authorize the client, you will get redirected to our client's /authorized resource, which will then redirect back to the /feed resource. Behind the scenes the OAuthClientFilter will get the authorized access token from the service and the /feed resource will finally retrieve and return the list of messages from the server.
8. If you hit refresh in the browser, you will see that further requests authorization is not necessary. That's because our client now has an authorized token which remains valid until the user explicitly revokes access on Twitter/MicroBlog pages. If we persisted the token, we could even use it after the restart of our application.

Step 7: Using JAXB and MVC Support in Jersey

The basic functionality is now implemented, but the output is not human readable – it is a raw XML that is sent from the service. In this section we are going to improve that. First, we will see how we can convert the XML we get from the service to Java objects, so that we can conveniently work with it further. Then we will use Jersey support for templates to render these Java objects into a formatted HTML output.

Jersey and JAX-RS has a notion of pluggable MessageBodyReaders and Writers, these are used by the framework to conver content of a particular set of media types to/from Java objects. Set of the default Readers/Writers that are shipped with Jersey support XML <-> Java conversion using JAXB. All we have to do to make it work is annotate the classes representing the data with the right JAXB annotations.

Let's examine how the XML output from Twitter/MicroBlog looks like. It is essentially an array/collection of „status"objects, which have various attributes. For our application, the most important attributes included in the XML are:

- created_at – indicate when the message was created
- text – text of the message
- user – object representing a user with the following attributes of interest:
    - name – user name

Let's create JAXB beans corresponding to the above data structures:

1. Right-click the com.mycompany.twitterwebclient package, choose New->Java Class and name it User.
2. Add a field and a getter corresponding to the name property. Attach the JAXB annotation @XmlElement to the name field to indicate this attribute is represented in the XML as an element.

```
public class User {
    @XmlElement(name = "name") String name;

    public String getName() {
        return name;
    }
}
```

3. Similarly, we'll add a class named Status. This class will have attributes corresponding to "created_at", "text", and "user" XML elements. Since the Status is the top-most Java class in the hierarchy corresponding to the message XML, we need to also annotate it with @XmlRootElement annotation. Here is how the whole class should look like:

```
@XmlRootElement
public class Status {
      @XmlElement(name = "created_at") String createdAt;
      @XmlElement(name = "text") String text;
      @XmlElement(name = "user") User user;

      public String getCreatedAt() {
            return createdAt;
      }

      public String getText() {
            return text;
      }

      public User getUser() {
            return user;
      }
}
```

4. The data objects are ready. Let's switch back to the FeedResource class. We are going to update the getIt() method to return a formatted list of the messages as an HTML page. As mentioned, Jersey has a nice support for templates which we are going to use for that. First we need to update the media type of the content the get method is going to produce. So, change the media type in the @Produces annotation to "text/html".
5. To make Jersey use the templating mechanism, the method is going to return an instance of a class named Viewable. It is a built-in Jersey class that the framework knows how to convert to an HTML document based on the supplied template. We can also change the name of the @GET method to anything we want (all that matters to Jersey is the @GET annotation) – so let's call it getFeed() instead of getIt(). The resulting method header will look as follows:

```
    @GET
    @Produces("text/html")
    public Viewable getFeed() {
        <method body>
    }
```

6. In the body of getFeed() (former getIt()) method, we can now change the code requesting the data from Twitter/MicroBlog in form of a String to request it as form of List<Status> - Jersey will take care of the conversion automatically. I.e. rewrite the following:

```
        // make an HTTP GET request to request a response of media type "application/xml"
        // and convert it to String
        String statuses = r.accept(MediaType.APPLICATION_XML).get(String.class);
```

As follows:

```
        // make an HTTP GET request to request a response of media type "application/xml"
        // and convert it to List<Status>
        List<Status> statuses = r.accept(MediaType.APPLICATION_XML)
                .get(new GenericType<List<Status>>() {});
```

Note we need to use GenericType, which is a helper class in Jersey to represent parameterized type. Ideally we would want to pass List<Status>.class to the get method, but that's not possible due to Java type erasure (the type the List is parameterized with does not appear in Java bytecode), so we have to work around it using the GenericType class.

7. Update the return statement to return Viewable instead of statuses as follows:

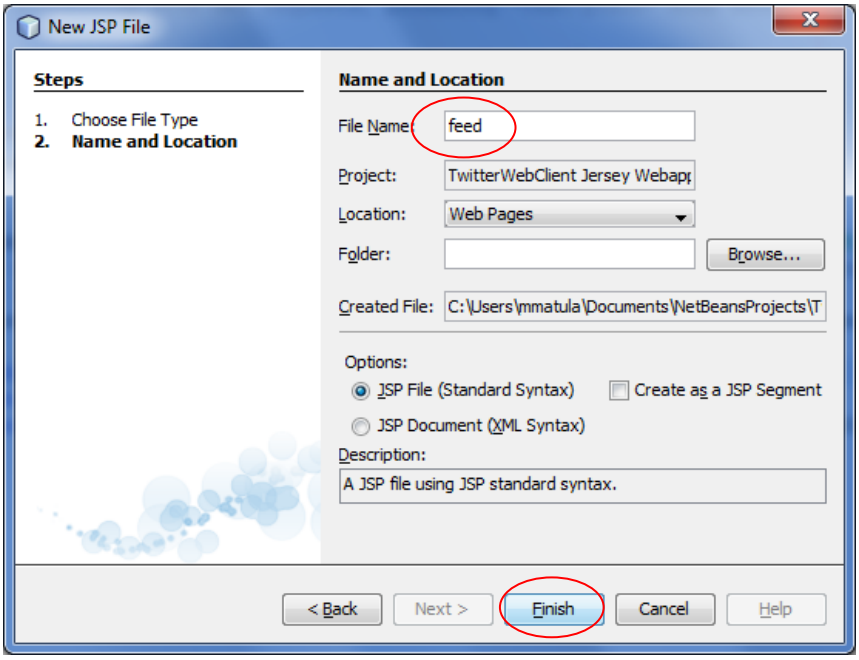
```
            return new Viewable("/feed", statuses);
```

Jersey supports JSP files as templates out of the box. Returning the above Viewable object tells Jersey to render the list of statuses using "/feed" template – i.e. feed.jsp file located in the root of the Web Pages folder of the project. Jersey will pass the content of the statuses variable to the jsp page as an attribute named "it".

8. Here is how the FeedResource class should look like after these changes:

```
@Path("/feed")
public class FeedResource {
    @GET
    @Produces("text/html")
    public Viewable getFeed() {
        try {
            // client resource representing Twitter/MicroBlog feed on the server
            WebResource r = Settings.getClient().resource(Settings.FRIENDS_TIMELINE_URL);
            // make an HTTP GET request to request a response of media type
            // "application/xml" and convert it to List<Status>
            List<Status> statuses = r.accept(MediaType.APPLICATION_XML)
                    .get(new GenericType<List<Status>>() {});
            // return the result
            return new Viewable("/feed", statuses);
        } catch (UnauthorizedRequestException e) {
            // throw WebApplicationException exception that is be automatically
            // mapped by Jersey to the specified response
            // (i.e. redirect to the URI obtained from e.getAuthorizationUri())
            throw new WebApplicationException(Response.seeOther(e.getAuthorizationUri())
                .build());
        }
    }
}
```

9. Let's create the feed.jsp page. Right-click the Web Pages node in the project explorer, and select New -> JSP, name it "feed" and click Finish.



10. In the generated page, update the text in the <title> tag to „OAuth Web Client – My Feed“ and the <h1> tag with „My Feed“.
11. We will use JSP Standard Tag Library tags to iterate through the list of statuses, so add the following At the top of the page, right after @page element:

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
```

12. Now we will add the code that will iterate through the list of statuses and format them on the page. Use <c:forEach> tag from JSTL to achieve this, together with the Status and User JAXB beans we implemented recently. Add the following code right after the <h1>…</h1> header in the body of the page.

```
<c:forEach var="status" items="${it}">
    <p>
    ${status.text}<br/>
    <small>posted by <i>${status.user.name}</i>
    at <i>${status.createdAt}</i></small><br/>
    </p>
</c:forEach>
```

13. And we’re done! The complete feed.jsp should look like this:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html">
        <title>OAuth Web Client – My Feed</title>
    </head>
    <body>
        <h1>My Feed</h1>
        <c:forEach var="status" items="${it}">
            <p>
            ${status.text}<br/>
            <small>posted by <i>${status.user.name}</i>
                    at <i>${status.createdAt}</i></small><br/>
            </p>
        </c:forEach>
    </body>
</html>
```

14. Run the project again. NetBeans will redeploy the application, so the access token will be lost and you'll have to go though the authorization again. Observe that the output is now formatted. You can play with the application by posting tweets on Twitter or messages on MicroBlog and refreshing the feeds page of our client in the browser to see it reads up-to-date data with no need for repeated authorization. You can try revoking access in Twitter/MicroBlog and observe that upon the next refresh of the feeds page, the client will ask for another authorization.

Summary

In this exercise we explained how OAuth works, and also touched on some more advanced Jersey features like JAXB integration and MVC (model-view-controller) architecture support. We used the information to build an OAuth enabled Twitter client application. For completeness, along with the solution for this exercise, we also included a command-line client in the lab zip file, so that you can look at it at home and explore the differences in the flow. In the next exercise we are going to look at how to add OAuth support to the server side.

Exercise 3: Enabling OAuth in an Existing Service

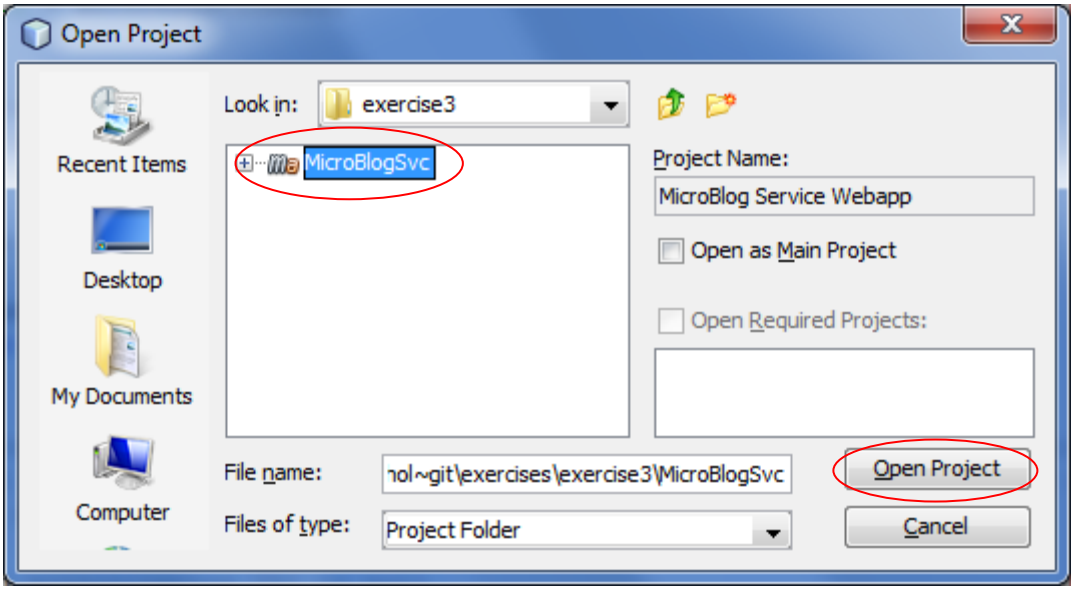
In this exercise, we will develop a subsset of the MicroBlog application used in the previous exercise as a service. As a starting point, we will use MicroBlog developed to the phase when the basic functinoality is in, but there is no OAuth support. All that the MicroBlog application does in this stage is the form-based authentication and user registration and allowing CRUD operations on the user messages. As part of the lab setup (see Install and Configure Lab Environment section) a user „hol-user“ with password „javaone“ was added to the security realm named „file“ – we can use these user credentials to log into the application through the web browser and create a few messages. Once we enable OAuth, we are going to use Firefox REST Client add-on to see if it works. Using this addon, you will see how exactly the OAuth-related communication looks like on the wire. And of course, you will be able to use the client you built in Exercise 2 to interact with this service as well.

Steps to Follow

Step 1: Exploring the Existing Web Application Project

Before we start writing any code, let’s explore the existing application to see how it is designed, run it and try to play with it for a while.

- In NetBeans menu go to File->Open Project, in the Open Project dialog browse to <lab_root>/exercises/exercise3 directory, select MicroBlogSvc project and click Open Project.



- Let’s try to build and run the web application project to see what it does. Right-click on the MicroBlog Service Webapp project and click Run.

- NetBeans will build and deploy the application, and start a web browser, so you should see the MicroBlog login form. Enter the following credentials and press Enter:

  User name: hol-user
  Password: javaone

- Once you are logged in, try adding, editing and deleting a few statuses/messages. Please, keep in mind this is just a demo application which does not handle some edge cases (e.g. escaping problematic characters such as quotes, greater than, etc.), so try to avoid these.

Let's take a quick look at how this is implemented:

- The application supports the following resources and operations on them:

Path (from the resource root)	**Matching Resource Class**	**Operation**	**Description**
/	RootResource	GET	Redirect's /{user}/notes/
/login	RootResource	GET	Enforces form-based authentication and redirect's back to the URL that was passed in the „redirect" query parameter.
/logout	RootResource	POST	Invalidates the session and redirects to the root („/").
/register	RootResource	GET	Retrieves an HTML form for the user to enter the registration info.
		POST	Registers a new user.
/{user}/statuses/	StatusesResource	GET	Retrieves a list of notes for a given user.
		POST	Adds a new note for a given user.
/{user}/statuses/new	StatusesResource	GET	Retrieves an HTML form to prompt for the data for a new note.
/{user}/statuses/{statusId}	StatusResource	GET	Retrieves a note by ID.
		POST	Updates a note.
		DELETE	Deletes a note.

- In web.xml there is the following security configuration:
  - Authentication required for <app_context_root>/login
  - Login configuration is set to Form (i.e. form-based authentication) with login.jsp as the Login Page and login−error.jsp as the Error Page, realm name set to „file".
  - One role is defined – user – which is mapped (in sun-web.xml) to a group named „users"

- For each request, the application checks if a user is logged in (i.e. user principal in the security context is not null). This check is done in the constructor of the UserResource class. If nobody is logged in, the application throws RequestProcessingException which is then mapped by the RequestProcessingExceptionMapper to a redirect to /login path. As mentnioned above, an authentication constraint is set on that path – so, by redirecting there, the container will trigger the form-based authentication using the „file“ realm (the one into which we added the „hol-user“ user during the lab setup – so we are able to use the „hol-user“ account to log in).
- For the requests that modify data, the application checks if the logged in user is in „user“ role. That is the case if the user has logged in through the form-based authentication and is a member of „users“ group in the file realm, which is mapped to „user“ role per the configuration in sun-web.xml. This is the case for the „hol-user“.
- The application uses Jersey MVC feature, which you had a chance to use in the previous exercise, so the GET methods on resources return Viewables, with the JSP page name in the parameter. The JSP pages mapping is as follows:

JSP	Method	Description
footer.jsp	-	Used as a footer in all JSP pages.
header.jsp	-	Used as a header in all JSP pages.
login-error.jsp	-	Loaded by the container when the form-based authentication fails (wrong credentials).
login.jsp	-	Loaded by the container when the form-based authentication is triggered – collects user credentials.
register.jsp	RootResource.registrationPage()	Registration HTML form.
newstatus.jsp	StatusesResource.getForm()	New note HTML form.
status.jsp	StatusResource.getHtml()	HTML representation of the note.
statuses.jsp	StatusesResource.getHtml()	HTML representation of the list of notes.

- The application uses JAXB beans for the statuses data – i.e. Status object is a JAXB bean, list of statuses is modeled as List<Status>. Since it is a JAXB bean, it can be used to un-/marshall Statuses from/to XML when sent accross the wire. Same for the User bean.
- To keep it simple, the application is not backed by any database. The statuses are stored in a static HashMap in the DataProvider class that also provides operations for statuses retrieval and updates.

Now that we briefly explored the application structure, we will look at how we can extend this application to allow one more type of authentication – OAuth.

Step 2: Designing the Changes to the MicroBlog to Implement OAuth

From the OAuth overview at the beginning of this lab we know our application will have to provide several additional facilities to support OAuth:

- Client (consumer) management – i.e. client registration/deletion, generating consumer keys and secrets
- Token management – generating request and access tokens and token secrets, providing a way for the user to authorize tokens, retrieve list of authorized tokens and revoke access to previously authorized clients
- Verification of the OAuth requests – each incoming request will need to be verified and if it contains a valid authorized token, a unique token/nonce/timestamp combination and is properly signed, we should set the user principal in the security context to the user who authorized that particular token. For the demo purposes, we will not set any role. Since we require „user“ role for write operations, this implies OAuth clients will only have read access.

The good news is, Jersey oauth-server library can help with all the things mentioned above. Most of what we will need to add is the web front end for the functonality that the oauth-server module provides. Here are the URI paths we are going to add to our application:

Path	Matching Resource Class	Operation	Description
/authorize	AuthorizationResource	GET	Authorization URL the client will have to redirect to, after getting the Request Token. Returns a confirmation HTML page asking user to confirm authorization of the client. Implements steps 3 and 4 of the OAuth authorization flow (see the diagram at the beginning of exercise 2).
		POST	Authorizes or denies (depending on the values of the form parameters) the passed request token and redirects to the client callback URL. Implements steps 5 and 6 of the OAuth authorization flow.
/{user}/applications/		POST	Registers a new client (consumer), redirects to the consumer resource: /{user}/applications/{consumerKey}
/{user}/applications/register	ConsumersResource	GET	Returns an HTML form for registering a new client.
/{user}/applications/{consumerKey}	ConsumerResource	GET	Returns an HTML representation of a client registration.

The oauth-server module provides an SPI you can implement to plug in your own implementation of token and consumer management. However, for the purpose of this lab we are going to leverage the default in-memory implementation which comes as part of the oauth-server module (mainly for the testing purposes).

OK, enough of exploring and designing. Let's get back to some coding.

Step 3: Configuring the Project to Use the Jersey oauth-server Library

1. Expand the MicroBlogSvc project node in the Project Explorer and add a dependency on oauth-server library by right−clicking on the Dependencies node, clicking Add Dependency filling it out as follows and clicking OK:

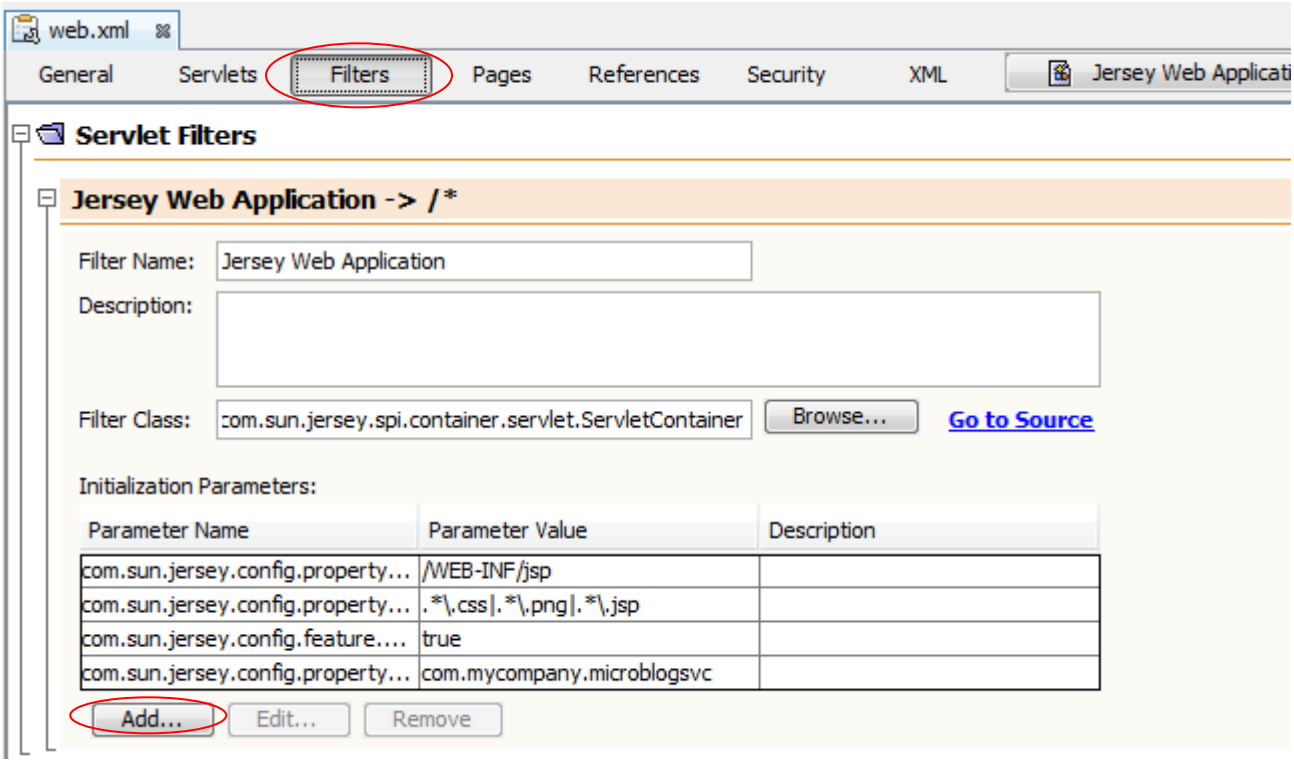
   GroupId: com.sun.jersey.contribs.jersey-oauth
   ArtifactId: oauth-server
   Version: ${jersey-version}

2. Do the same for oauth-signature library:

   GroupId: com.sun.jersey.contribs.jersey-oauth
   ArtifactId: oauth-signature
   Version: ${jersey-version}

3. Now, let's edit the web.xml file to configure the Jersey OAuth server filter and the OAuth provider implementation that should be used. Under the MicroBlogSvc project expand Web Pages->WEB-INF and double-click on web.xml to open it in a visual editor.

4. In the editor click on the Filters tab and under the Initialization Parameters section click Add.



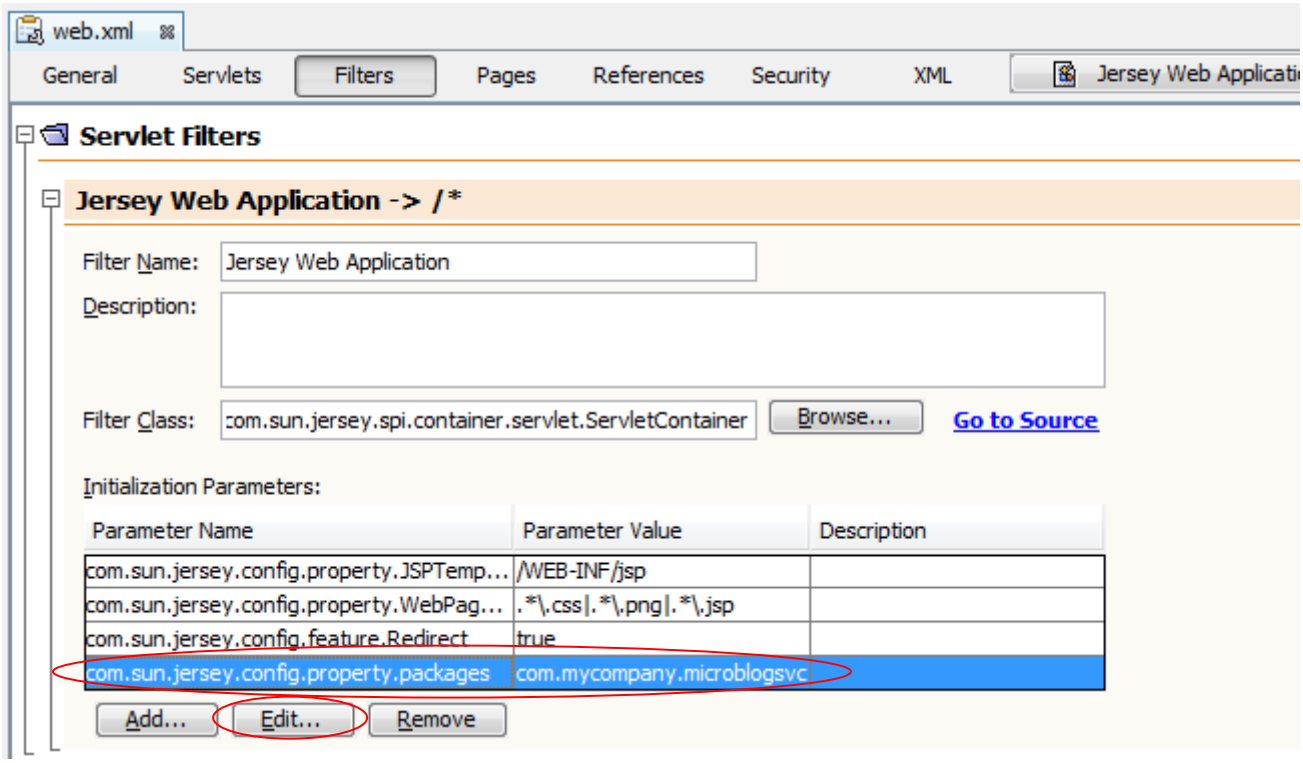
5. Fill out the fields in the Add Initialization Parameter dialog as follows and click OK.

   Parameter Name: com.sun.jersey.spi.container.ContainerRequestFilters
   Parameter Value: com.sun.jersey.oauth.server.api.OAuthServerFilter

   This registers the OAuth server filter provided by the oauth-server module as the container request filter – i.e. all requests will get processed by this filter before they reach our resources. The filter takes care of validating the request headers to confirm they have correct OAuth parameter values and proper signature and if so, sets the corresponding principal and roles to the security context.

6. The filter registered in the previous step needs to have a way of getting the information on consumers and tokens, to retrieve the consumer and token secrets and verify the consumer key and token included in the request are valid. For that it uses an OAuthProvider implementation the service provider developer has to register through the initialization parameters as well. As we mentioned, we are going to use the default OAuthProvider implementation provided by the oauth-server module. This provider is in the com.sun.jersey.oauth.server.api.providers package. All we have to do to register it is let Jersey know it should include this package when searching for providers and resources. We will do this in the next step.

7. The oauth-server module also provides RequestTokenResource and AccessTokenResource implementing handling of the request token and access token requests. We want Jersey to also pick up these (along with the default OAuthProvider implementation). These are in com.sun.jersey.oauth.server.api.resources package. So, we will configure Jersey to look into their common „parent“package – com.lsun.jersey.oauth.server.api. In the web.xml editor, in the Initialization Parameters table click on the line with com.sun.jersey.config.property.packages parameter name (you may need to resize the column width to be able to see the whole parameter name) and click the edit button.



8. This parameter lists the names of packages where Jersey should look for the resource classes. We need to append com.sun.jersey.oauth.server.api to that list, so the resulting parameter value should look as follows:

   Parameter Name (unchanged): com.sun.jersey.config.property.packages
   Parameter Value: com.mycompany.microblogsvc,com.sun.jersey.oauth.server.api

   Once you change the value, click OK in the Edit Initialization Parameter dialog.

9. Finally, we need to tell the OAuth server filter to ignore the requests to RequestTokenResource and AccessTokenResource as well as the authorization, login and registration URLs, since are not to be accessed by an OAuth service or serve for obtaining an authorized token and until that happens the consumer can't send authorized OAuth headers. This can be achieved by adding the following Initialization Parameter:

   Parameter Name: com.sun.jersey.config.property.oauth.ignorePathPattern
   Parameter Value: requestToken|authorize|accessToken|register|login|logout

   So click Add under the Initialization Parameters table add the parameter above and click OK.

At this point, our application is configured to verify OAuth requests and exposes request token and access token resources described in the following table:

Path	HTTP Method	Consumes	Produces
/requestToken	POST	application/x-www-form-urlencoded	application/x-www-form-urlencoded
/accessToken	POST	application/x-www-form-urlencoded	application/x-www-form-urlencoded

Now we need to add a web interface for registering new consumers/clients.

Step 4: Adding Consumer Management/Registration

1. Create a new class in the com.mycompany.microblogsvc package, name it ConsumersResource. This will be a resource for managing consumers.
2. Make the class extend ResourceBase, and set the URI template to „/{user}/applications/".

```
package com.mycompany.microblogsvc;

import javax.ws.rs.Path;

@Path("/{user}/applications/")
public class ConsumersResource extends ResourceBase {
}
```

3. Add an HTML GET method producing text/html at the „register" sub-path. The method should return a page with an HTML form to enter information about the consumer being registered. We will do this by returning Viewable, referencing a JSP page named consummerregistration.jsp.

```
    @Path("register")
    @GET
    @Produces(MediaType.TEXT_HTML)
    public Viewable getHtml() {
        // make sure the right user is logged in
        checkUser();
        return new Viewable("/consumerregistration", null);
    }
```

NOTE: When fixing imports for the above code, make sure to import MediaType from package javax.ws.rs.core.

4. Now we need to add a POST method that will be called when a user submits the form returned from the method above. This POST method will register a new consumer, and redirect to its detail view. The consumer registration will be performed by calling registerConsumer() method on the DefaultOAuthProvider class (implementation of OAuthProvider) that we registered in the previous steps. This class is provided by the oauth-server module and uses static hash maps to store all the information about tokens and consumers. To get a reference to the singleton instance of this class, we can inject it into a field of our class using the @Context annotation.

```
    // inject the DefaultOAuthProvider instance
    private @Context DefaultOAuthProvider provider;

    @POST
    @Produces(MediaType.TEXT_PLAIN)
    public Response post(Form params) {
        // make sure the right user is logged in
        checkUser();
        // only allow if the user has write access
        checkWrite();
        // register a new consumer
        Consumer consumer = provider.registerConsumer(user, params);
        // redirect to the consumer detail
        return Response.seeOther(uriInfo.getRequestUriBuilder()
                .path("{consumerKey}").build(consumer.getKey())).build();
    }
```

NOTE: When fixing imports after copy-pasting this block of code, make sure you import javax.ws.rs.core.Context and for Form, make sure to import com.sun.jersey.api.representation.Form rather than Form from some other package.

As you can see, this method takes Form as an argument. Form is a class corresponding to a set of HTML form parameters and values. It will be populated automatically by the Jersey framework with the form parameters posted by the consumer registration form. Passing the whole form object to the registerConsumer method of the default OAuthProvider implementation results in storing all form parameters as the custom consumer attributes. So, whatever form parameters we will come up with (like consumer name, URI, etc.) will be stored in the consumer object.

5. The last method we will add to the ConsumersResource for now is the GET method that returns Viewable corresponding to the consumer detail view. We will use „{consumerKey}“ sub-path for this resource.

```
    @Path("{consumerKey}")
    @GET
    @Produces(MediaType.TEXT_HTML)
    public Viewable getConsumer(@PathParam("consumerKey") String consumerKey) {
        checkUser();
        // retrieve the consumer, check the owner and pass it to the viewable
        DefaultOAuthProvider.Consumer c = provider.getConsumer(consumerKey);
        if (!user.equals(c.getOwner())) throw new NotFoundException();
        return new Viewable("/consumer", c);
    }
```

6. So much for the Java code for this. Now you need to add the JSP pages that are used by the two GET methods to render the data. First add a new JSP named „consumerregistration“ under Web Pages/WEB-INF/jsp/. This JSP should contain a form for collecting the info about the consumer and posting the result to the parent path. For simplicity, we will collect only one piece of information about a consumer – its name. The JSP also needs to include the header.jsp and footer.jsp (all of the Notebook jsp's include this to have the same look). So, the resulting JSP should look like this:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=MacRoman">
        <link rel="stylesheet" type="text/css"
href="${pageContext.request.contextPath}/style.css" />
        <title>Consumer Registration</title>
    </head>
    <body>
        <%@include file="header.jsp"%>
        <h1>Consumer Registration</h1>
        <form action="." method="post">
            <table><tr>
                    <td>Consumer name:</td>
                    <td><input type="text" name="name"/></td>
                </tr><tr>
                    <td/>
                    <td>
                        <button type="submit">OK</button>
                        <button type="button" onclick="window.back()">Cancel</button>
                    </td>
            </tr></table>
        </form>
        <%@include file="footer.jsp"%>
    </body>
</html>
```

7. And add the second JSP for rendering the consumer detail. Name it „consumer“ (under Web Pages/WEB-INF/jsp/). Based on the ConsumersResource.getConsumer() method implementation, this page receives the Consumer object as a parameter (injected into an attribute named „it“ as you know from the previous exercise). So, here is how it should look like:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=MacRoman">
        <title>Application: ${it.attributes["name"][0]}</title>
        <link rel="stylesheet" type="text/css"
href="${pageContext.request.contextPath}/style.css" />
    </head>
    <body>
        <%@include file="header.jsp"%>
        <h1>Application Info</h1>
        <table>
            <tr><td>Application Name:</td><td><b>${it.attributes["name"][0]}</b></tr>
            <tr><td>Consumer Key:</td><td><b>${it.key}</b></td></tr>
            <tr><td>Consumer Secret:</td><td><b>${it.secret}</b></td></tr>
            <tr>
                <td></td>
                <td>
                    <button type="button"
                   onclick="location.href='${pageContext.request.contextPath}'">OK</button>
                </td>
            </tr>
        </table>
        <%@include file="footer.jsp"%>
    </body>
</html>
```

NOTE: We need to refer to the „name“ attribute of the consumer using a quite complicated construct (it.attributes[„name“][0]) which corresponds to consumer.getAttributes().get(„name“).get(0). This is because consumer attributes is of type Map<String, List<String>>, so, getAttributes().get(„name“) gets a list of all values for attribute „name“ for the given consumer, and since we know there will be just one value, we can directly call get(0) on that list.

Well done, we have just added the functionality to register consumers and show a consumer detail. The last remaining thing that needs to be done before we can use the OAuth mechanism in our application is implementing the token authorization.

Step 5: Implementing Token Authorization

If you look back at the OAuth description we provided in Exercise 2, we have the following steps covered:

1. Registering a consumer (done throught the resources we developed in the last step)
2. RequestToken request handling (done by the RequestTokenResource implemented by the oauth-server module)
3. AccessToken request handling (done by the AccessTokenResource implemented by the oauth-server module)
4. Handling of authorized requests (done by the OAuthServerFilter provided by the oauth-server module)

The last piece missing is implementing the functionality to allow users (resource owners) authorize the request tokens of the consumers, so that they can be exchanged for access tokens. So, the step that needs to happen between bullets 2 and 3 above.

Let's implement it by following these steps:

1. Add a new class named AuthorizationResource to com.mycompany.microblogsvc package, make it extend ResourceBase, attach the @Path annotation to set its path to „/authorize" and inject DefaultOAuthProvider into a private field named „provider".

```
@Path("/authorize")
public class AuthorizationResource extends ResourceBase {
    private @Context DefaultOAuthProvider provider;
}
```

   This resource will need to handle GET requests – this is where the consumer will redirect the user to authorize the consumer's request token – by asking the user to confirm the token authorization. Then it will also need to handle the POST requests triggered when the user chooses whether to approve or deny and submits. The GET request will only return the confirmation form, while the POST request handler will implement the logic of authorizing the token.

2. Add a method to handle GET requests, name it getHtml, make it return Viewable, produce text/html media type and accept oauth_token as the query parameter (consumer has to pass the request token to be authorized in the query parameter to this resource).

```
    @GET
    @Produces(MediaType.TEXT_HTML)
    public Viewable getHtml(@QueryParam("oauth_token") String token) {
        return new Viewable("/authorize", provider.getRequestToken(token));
    }
```

3. Add a new JSP named „authorize" under Web Pages/WEB-INF/jsp – this will be the one the above Viewable be rendered by – the token object is passed into it as the parameter. The JSP should provide a form informing user which consumer is requesting access and allowing the user to authorize. Let's make the JSP look as follows:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=MacRoman">
    <link rel="stylesheet" type="text/css"
href="${pageContext.request.contextPath}/style.css" />
    <title>Authorize</title>
  </head>
  <body>
    <%@include file="header.jsp"%>
    <h1>Authorization</h1>
    Please confirm that application "${it.consumer.attributes["name"][0]}" can
    get a readonly access to your account.
    <p>
    <form action="authorize" method="post">
      <input type="hidden" name="token" value="${it.token}"/>
      <table>
        <tr><td><input type="radio" name="approve" value="true" checked>Approve</td></tr>
        <tr><td><input type="radio" name="approve" value="false">Reject</td></tr>
        <tr><td><button type="submit">OK</button></td></tr>
      </table>
    </form>
    <%@include file="footer.jsp"%>
  </body>
</html>
```

As you can see, the approval confirmation is implemented using two radio buttons – Approve and Reject. Based on which one is selected, „approve“ form parameter is either „true“ (if approved) or „false“ (if rejected). The token string is passed in the hidden form parameter named „token“. We will implement the POST request handing accordingly.

4. Switch back to the AuthorizationResource class in the NetBeans editor and add a method handling HTTP POST requests named „post“ taking „token“ and „approve“ form parameters, and returning a Response object.

```
    @POST
    public Response post(@FormParam("approve") boolean approved, @FormParam("token") String
token) {
    }
```

5. Implement the method so that it authorizes the token if approved == true (by calling provider.authorizeToken()). Then it checks if the callback URL associated with the request token is null or set to OOB (out-of-bound) – this is the case when the callback URL is not applicable since the consumer is a desktop (not web) application or for some other applications this may also mean the callback URL was established by other means (e.g. during the consumer registration like in the case of Twitter in exercise 2). If so, our method should return a response containing the verifier returned from the token authorization, so that user can copy-paste it to the consumer application. Otherwise we should return a redirect to that callback URL providing the token string and the verifier in the query parameters. This is how the whole method should look like:

```
    @POST
    public Response post(@FormParam("approve") boolean approved, @FormParam("token") String
token) {
        // only allow if the user has write access
        checkWrite();
        DefaultOAuthProvider.Token t = provider.getRequestToken(token);
        if (t == null) {
            throw new NotFoundException();
        }

        String verifier = "";
        if (approved) {
            verifier = provider.authorizeToken(t, security.getUserPrincipal(), null);
        }

        String callback = t.getCallbackUrl();
        if (callback == null || callback.equalsIgnoreCase("oob")) {
            // return a plain text response with the verifier
            return
Response.ok("Switch back to your application and enter this verification code: " +
verifier).type(MediaType.TEXT_PLAIN).build();
        } else {
            // redirect to the callback URL providing the token and verifier in the params.
            URI uri = UriBuilder.fromUri(t.getCallbackUrl())
                    .queryParam(OAuthParameters.TOKEN, token)
                    .queryParam(OAuthParameters.VERIFIER, verifier).build();
            return Response.seeOther(uri).build();
        }
    }
```

6. Finally we need to provide a way for a user to get to this functionality – i.e. add a link to the consumer registration into the “link bar” in the header.jsp. Insert this at line 39 of header.jsp (i.e. right after the line starting with “<c:set “):

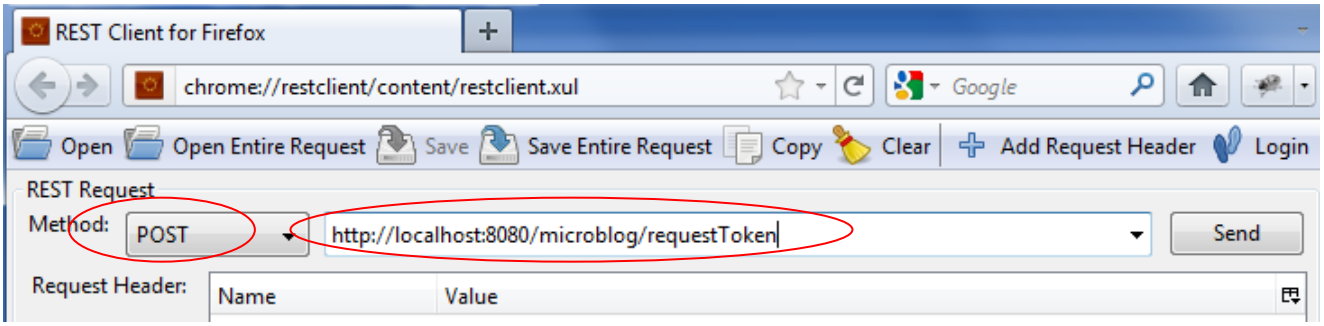
```
            <div class="right">
                <a class="links" href="${pathPrefix}applications/register">Register App</a>
            </div>
```

Step 6: Testing the OAuth Service Provider

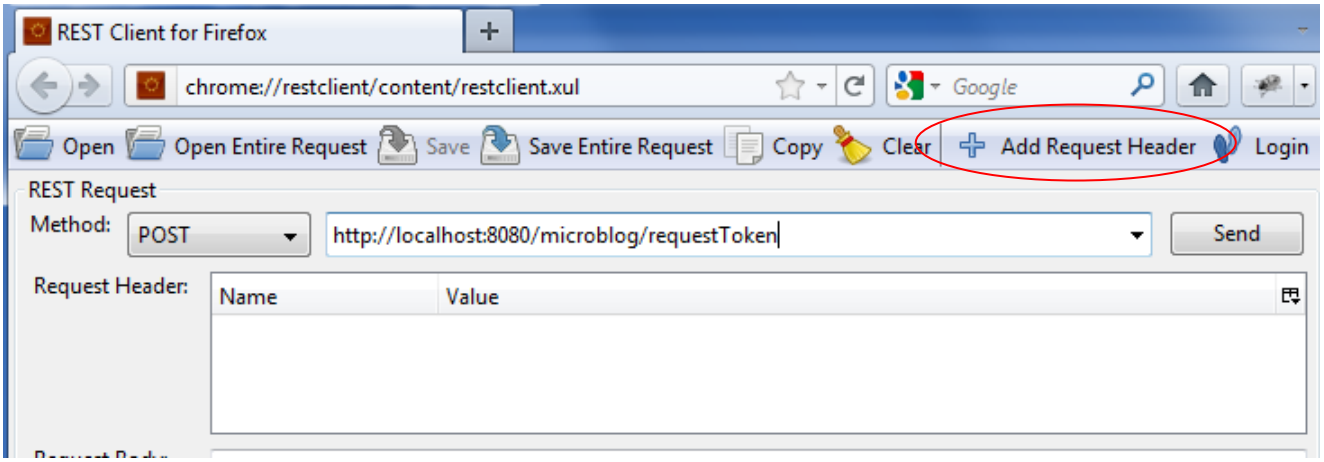
So, now that we are done making changes to enable OAuth in the MicroBlogSvc application, let’s run it and try it out.

1. Right-click on the MicroBlog Service Webapp and click Run in the pop-up menu. A Firefox browser window with the login screen to our MicroBlog application should pop up.
2. Log into MicroBlog (user: hol-user, password: javaone).
3. Click on the Register App link on the right, enter a name of the new consumer (e.g. „Test“) and click OK. Now you will see the new consumer details – including the consumer key and consumer secret.
4. Open a new Firefox window by pressing Ctrl+N (Cmd+N on a Mac) or clicking File->New Window in the main menu.

5. In this new Firefox window click Tools->REST Client in the main menu.This will open a tab with the REST Client add-on UI. This add-on allows us to construct HTTP requests based on our needs and also has support for OAuth signing of messages.
6. First we will configure the add-on to send a POST request to the request token URL to obtain a token. To do that, set the Method to POST and set URL to the request token URL (i.e. http://localhost:8080/microblog/requestToken).

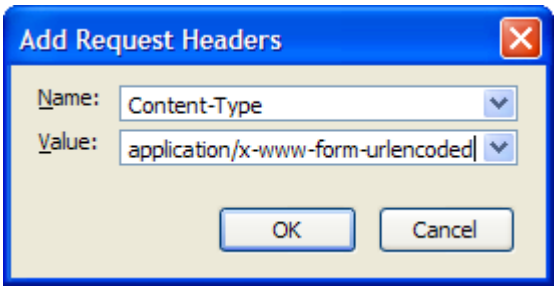


7. You also need to set the Content Type of the request in the header to application/x-www-form-urlencoded since only that media type is accepted for the request token requests. To do this, click Add Request Header in the toolbar of the REST Client tab.

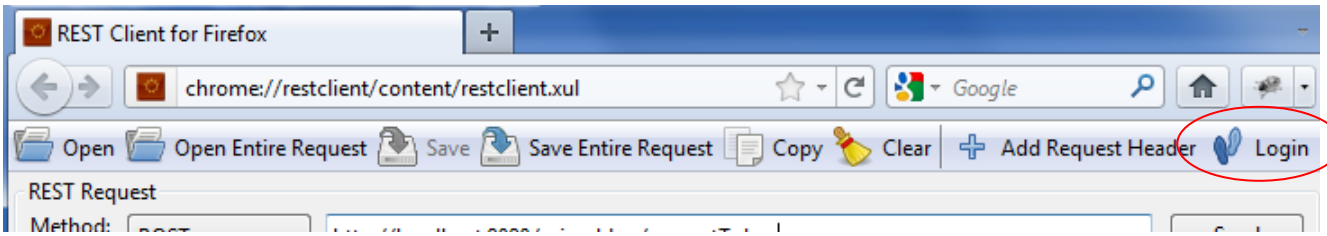


8. Fill the following values in the Add Request Headers dialog and click OK:

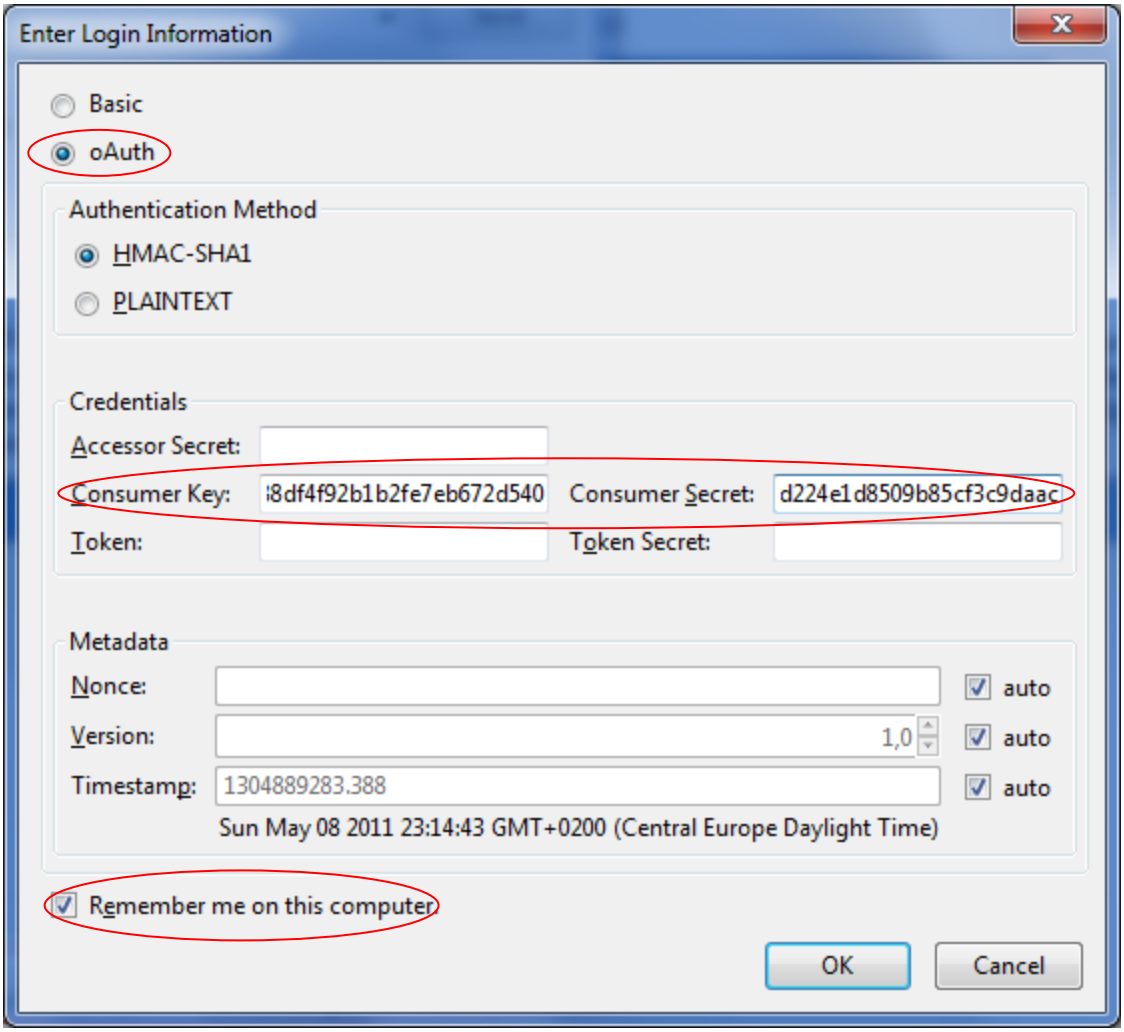
   Name: Content-Type
   Value: application/x-www-form-urlencoded



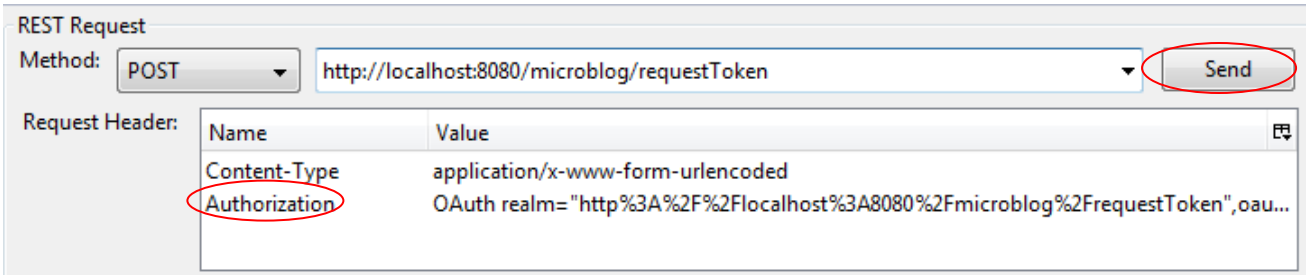
9. Now you have to share the consumer key and the consumer secret with the REST Client add-on, so that it can properly sign the message. To do this, click Login in the REST Client tab toolbar.



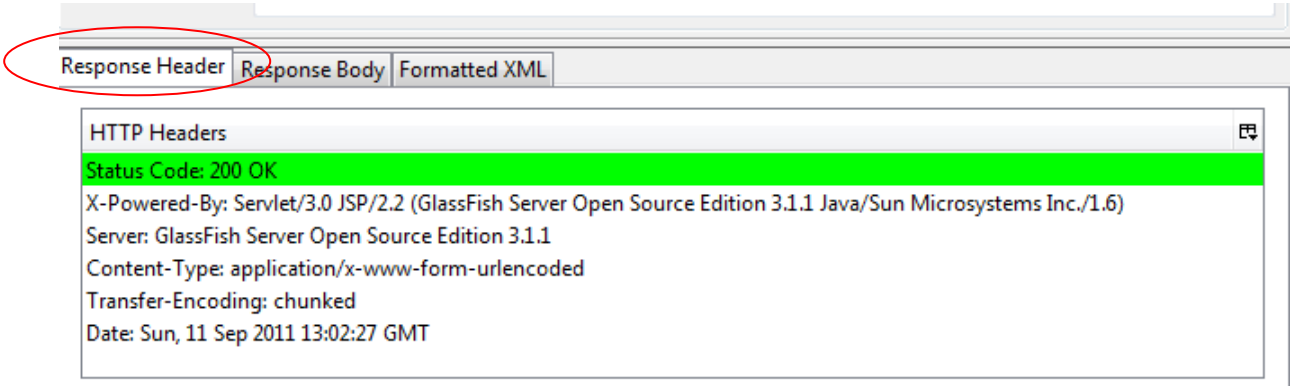
10. Dialog for entering the login information will appear. If the dialog is too small, resize it to see all the fields. Choose OAuth login method, check Remember me on this computer at the bottom of the window and copy-paste the consumer key and the consumer secret from the Notebook web app in the other Firefox window to the corresponding fields of the REST Client login window. Make sure the Token and Token Secret fields are empty and click OK.



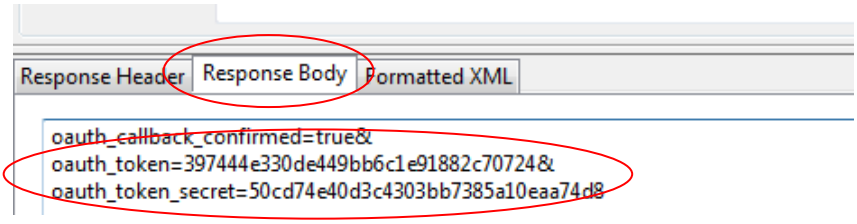
11. Now you should see the Authorization field added to the Request Header table. Send the request by clicking on the Send button.



12. If you set up everything correctly, at the bottom of the REST client window you should see the request returned status code 200 (OK).



13. Click on the Response Body tab at the bottom. That should contain the request token attributes (i.e. the token string and the token secret).



14. So, we received the request token. Now we need to get the user to authorize it. At this point a consumer would redirect to the authorization URL providing the token in a query parameter. So, let's switch back to the Firefox window with the MicroBlog application, and type in the following URL:

    http://localhost:8080/microblog/authorize?oauth_token=<copy-paste the returned token>

NOTE: As the text above suggests, you need to add a concrete oauth_token value to that URL which you can copy-paste from the Response Body of the request token request. In case of the above screenshot, the real URL you would type in is:

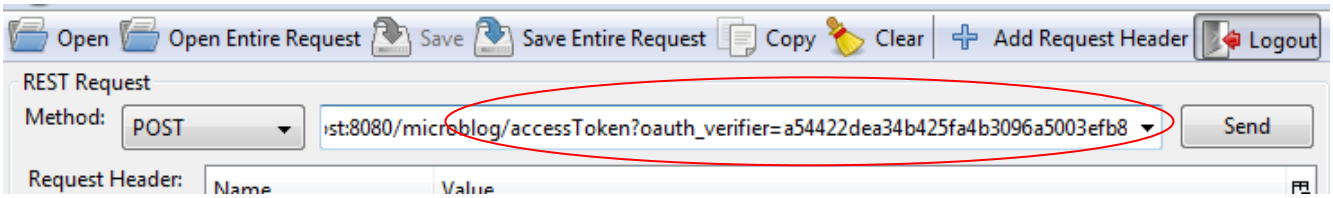
http://localhost:8080/microblog/authorize?oauth_token=397444e330de449bb6c1e91882c70724

15. Hit enter. The browser should show the authorization page asking you to either Approve or Reject access to the Consumer. Make sure Approve is selected and click OK.

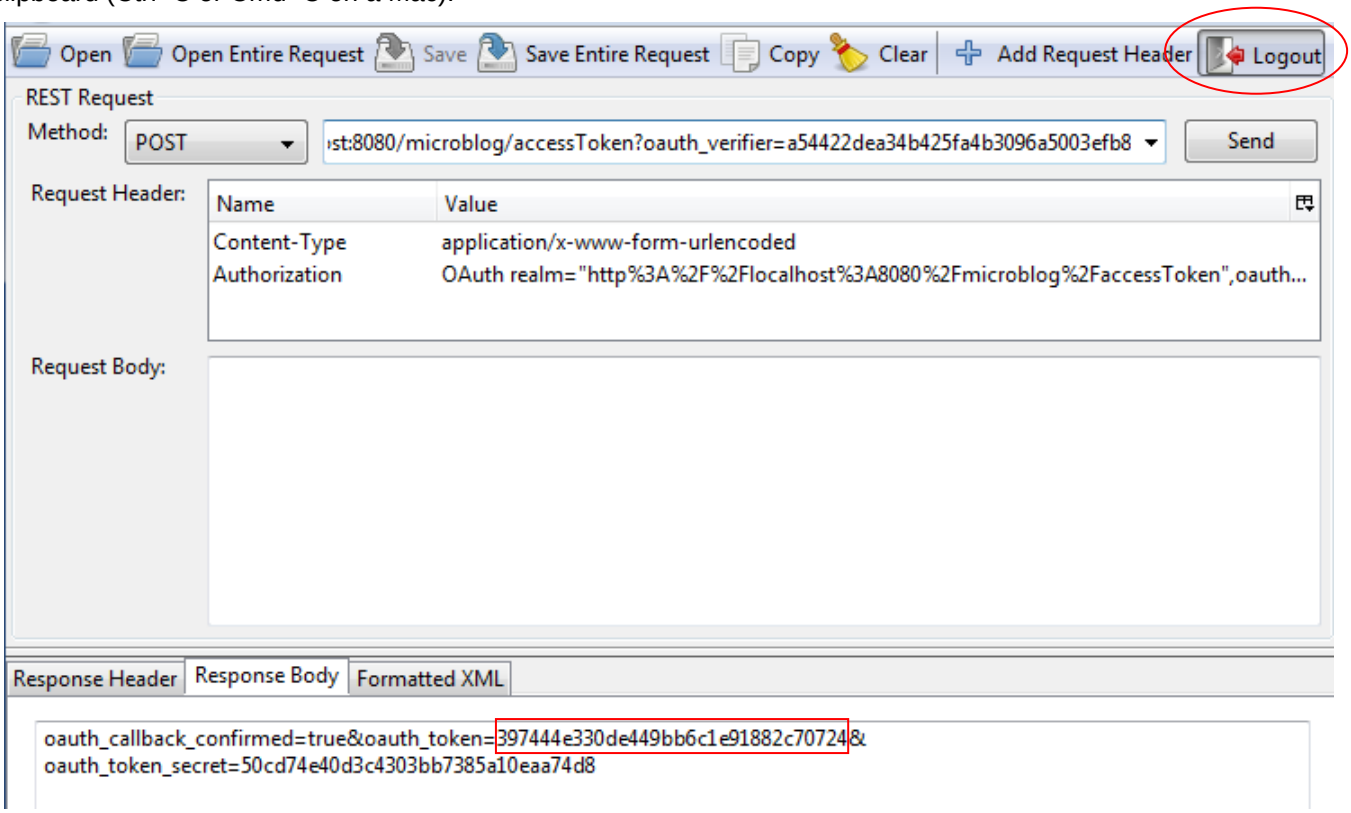
NOTE: Before getting to the authorization page, you may first get the login page in case the user session expired in the meantime. In that case just log in and you will get to the authorization page.

16. If the authorization goes through, a message with the verification code will be returned. This is because we have not set any callback URL when making the request token request. You will need to send this verification code in the access token request in the next step.

17. Switch back to the Firefox window with REST Client tab open. Replace requestToken in the request URL by accessToken and append a query parameter oauth_verifier set to the value of the verification code.



18. Now you need to add the request token string and secret to the OAuth login information. Unfortunately the login dialog is modal, so this will be a bit painful – we will have to log in and log out twice to copy-paste both values. So, click Logout button in the REST Client toolbar, copy the token string from the Response Body tab at the bottom to the clipboard (Ctrl+C or Cmd+C on a Mac).

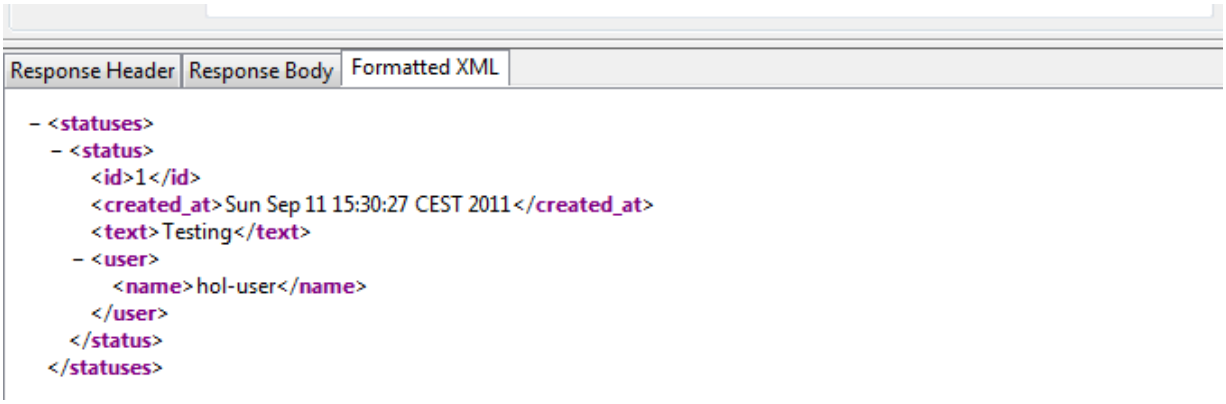


19. Click on the Login button, paste the token value to the Token field and press OK.
20. Click Logout again and copy the token secret value from Response Body tab to the clipboard.
21. Click Login again and paste the token secret value to the Token Secret field and press OK.
22. Send the access token request by clicking the Send button.
23. If you followed the previous steps correctly, you should now see a new token and token secret. This is the authorized access token you can now use in all subsequent requests. But first you need to set it to the OAuth login information by repeating the process you did for setting the request token (i.e. steps 18 – 21 above). Please do that.
24. Now that the access token is set in the header, try to make a request to retrieve a list of statuses for the user that authorized the token (hol-user in this case). First click „Clear“button in the REST Client toolbar – this will clean up the request URL, method and request headers.
25. Make sure the request method is set to GET and set the request URL to:

    http://localhost:8080/microblog/hol-user/statuses/
26. Besides HTML (which is not conveniently machine readable) the StatusesResource can return also application/xml and application/json representations of the list of statuses, which is more suitable for client applications, so let's set the Accept header to application/xml by clicking on Add Request Header and filling in the following values in the dialog:

    Name: Accept
    Value: application/xml

    Click OK.
27. Click the Logout button, then Login again and OK in the login dialog to make REST Client add the OAuth header.
28. Make a request by clicking Send. You should see an XML output in the Response Body listing the notes. For nicely formatted output switch to Response Body with Syntax Highlight tab.



NOTE: Since the notes are kept in memory only, there will be no notes (unless you played with the MicroBlog application after the last build and redeploy) when you make this reqeust for the first time. Create some notes in the Notebook application and try again.

29. You can now play with the notes through the web interface and then repeat the GET request on the notes resource in the REST Client (just by clicking Send again) to see that it reflects the current state.

Summary

In this exercise you saw how to use the oauth-server library to add OAuth support to your service and how to try it out using a generic client. You added just a basic functionality for registering the consumers. The lab zip file also provides a more complete implementation of MicroBlog service in the extras folder of the lab zip file. We encourage you look at the sources to see how to add more features such as ability to revoke access to a particular client, etc.

Summary

Congratulations! You have successfully completed this lab. You have learned how to use JAX-RS, Jersey and its extensions to build RESTful web services utilizing OAuth for authorization, and how to build clients to OAuth enabled services.

For additional information about the technologies used in this lab, please see the following links:

- http://oauth.net/ - OAuth web site
- http://jersey.java.net/ – Project Jersey web site
- http://wikis.sun.com/display/Jersey – Jersey wiki
- http://dev.twitter.com/ - Twitter API pages

If you have questions or comments about the lab, you can post them on the Jersey mailing list here:

users@jersey.java.net

You can also contact the lab authors directly at:

Martin.Matula@oracle.com

Pavel.Bucek@oracle.com

Thank you for participating!

Appendix: Setting Up the Lab Environment

System Requirements

- Supported operating environments: any that can run the required software mentioned below
- Memory requirements: 2GB
- Disk spare requirements: 2GB

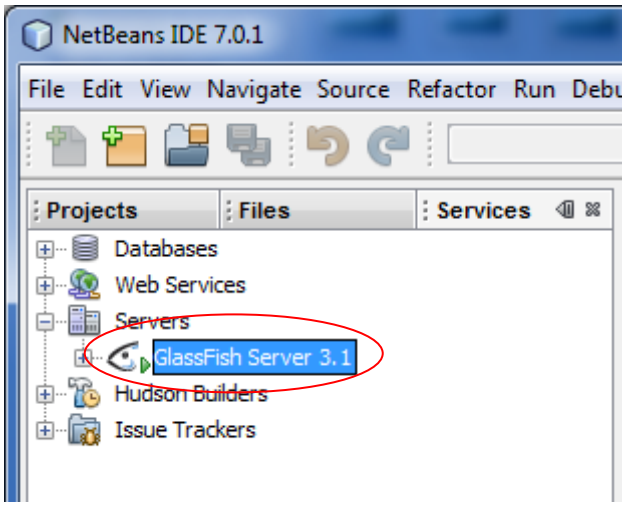
Software Needed For This Lab

Please install the following set of software:

- Java SE Development Kit 6
    - Make sure JAVA_HOME environment variable is set and points to your JDK installation
- Firefox 3.5 or more recent
- REST Client 1.3.1 (or more recent) Firefox Add-on
- NetBeans 7.0.1 (Java EE or All bundle that includes GlassFish 3.1.1)

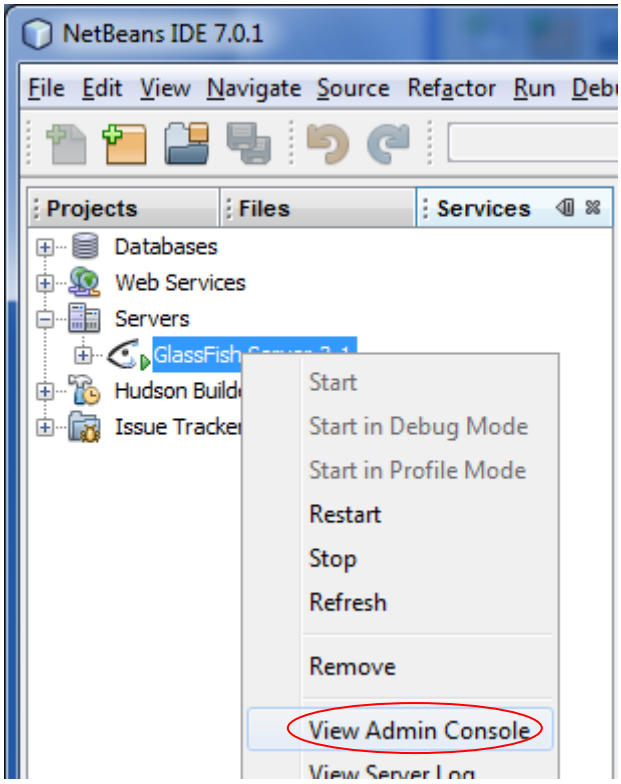
Install and Configure Lab Environment

- Download and unzip the lab zip file.
- Configure GlassFish (using NetBeans)
    1. Switch to the Services tab in the project explorer and expand the Servers node. You should see GlassFish Server 3.1.

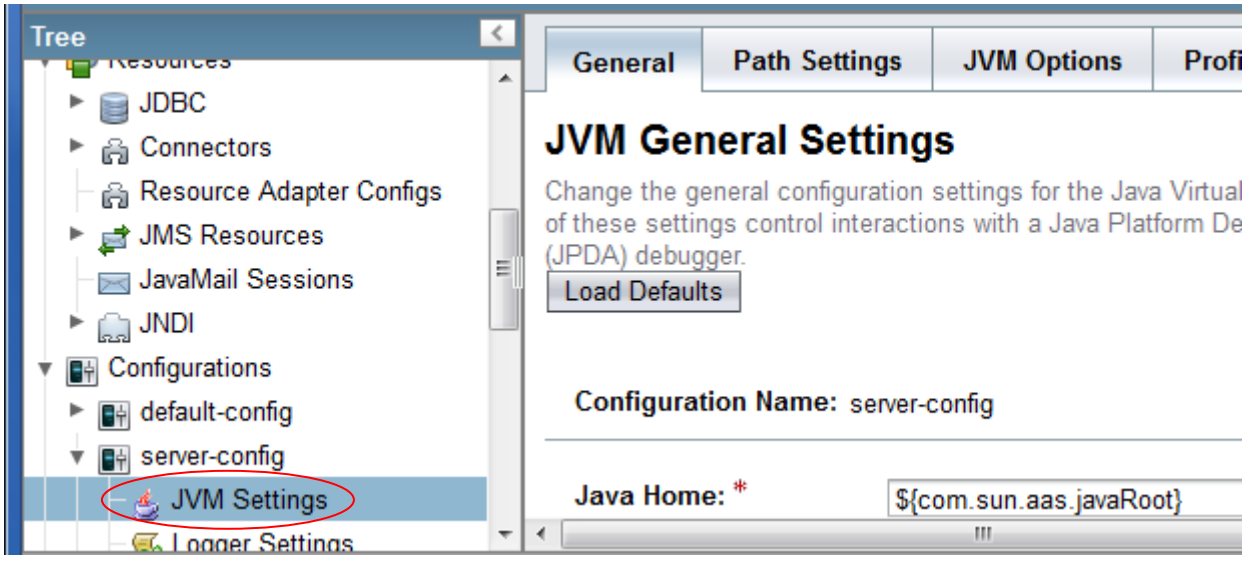


    2. Start the GlassFish server by right-clicking on it and choosing Start from the pop-up menu.

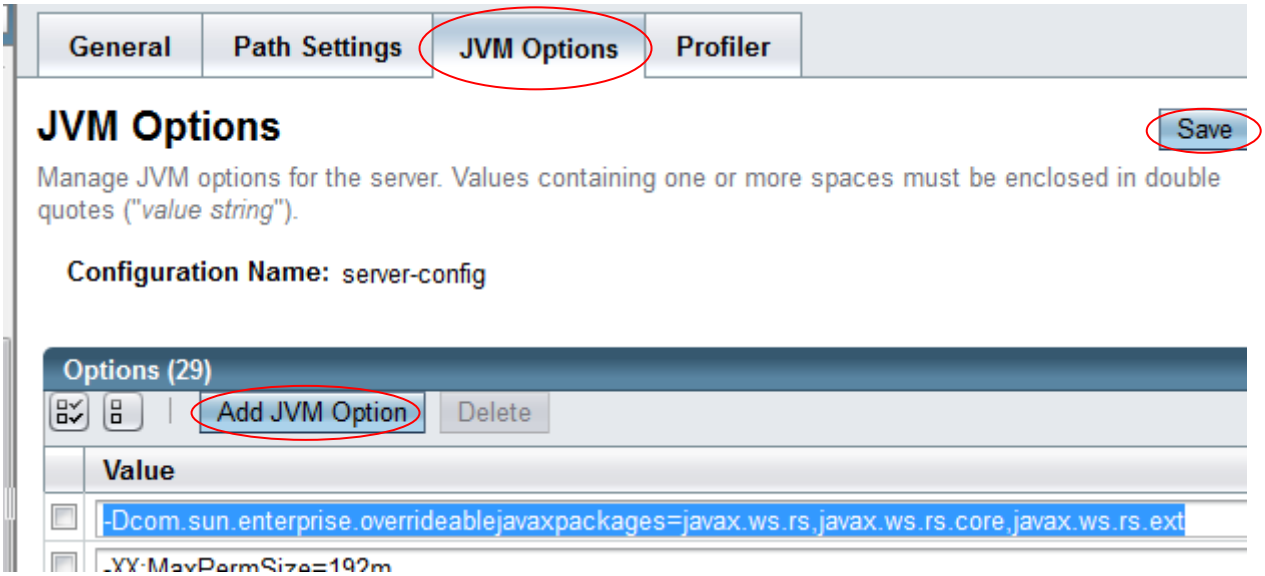
3. Once the server is running (small green rectangle appears in the bottom-right corner of its icon), start the Admin Console by right-clicking on the server node again and choosing View Admin Console from the pop-up menu.



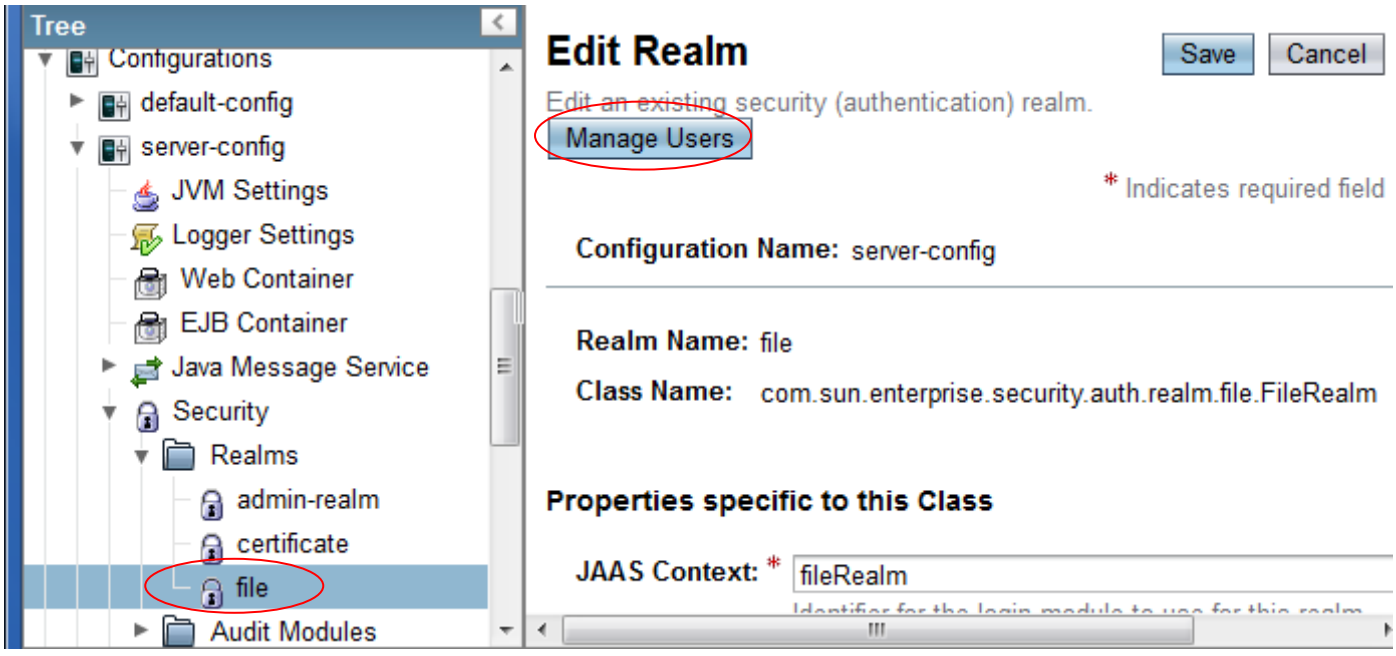
4. Once the Admin Console shows up in a web browser window, click Configurations->server-config->JVM Settings in the treeview on the left.



5. Switch to the JVM Options tab and click Add JVM Option. An empty field will be added to the top of the list. Type -Dcom.sun.enterprise.overrideablejavaxpackages=javax.ws.rs,javax.ws.rs.core,javax.ws.rs.ext into it and click Save. Setting this property will let us override the version of Jersey that's built into GlassFish, as we need to use a newer version for this lab.



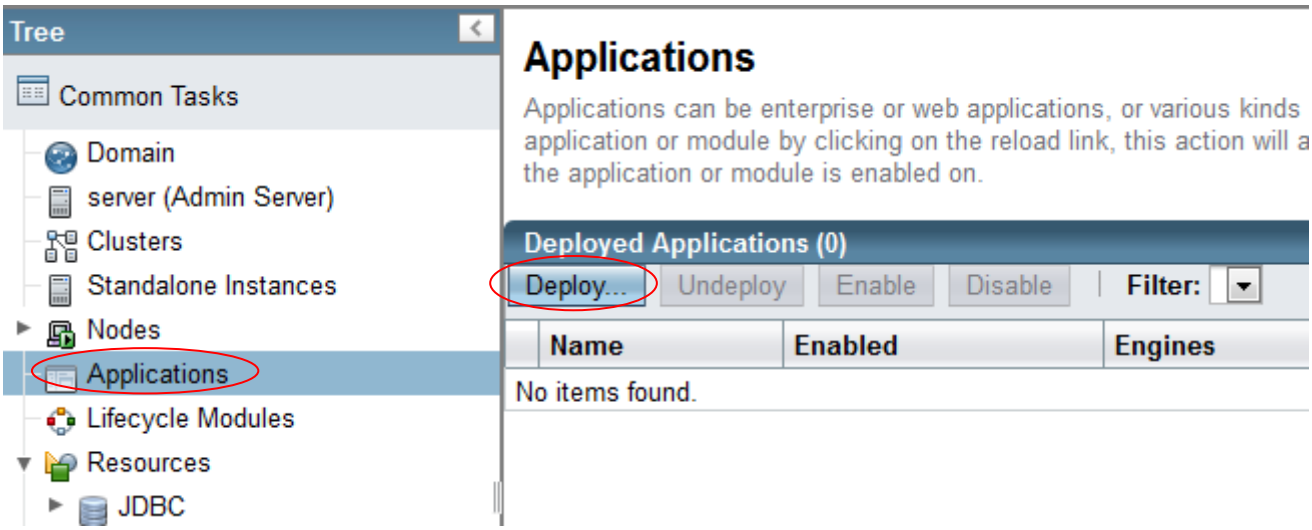
6. Now in the tree view on the left, expand Configurations->server-config->Security->Realms, and click on „file" under Realms. Then click Manage Users in the window pane on the right.



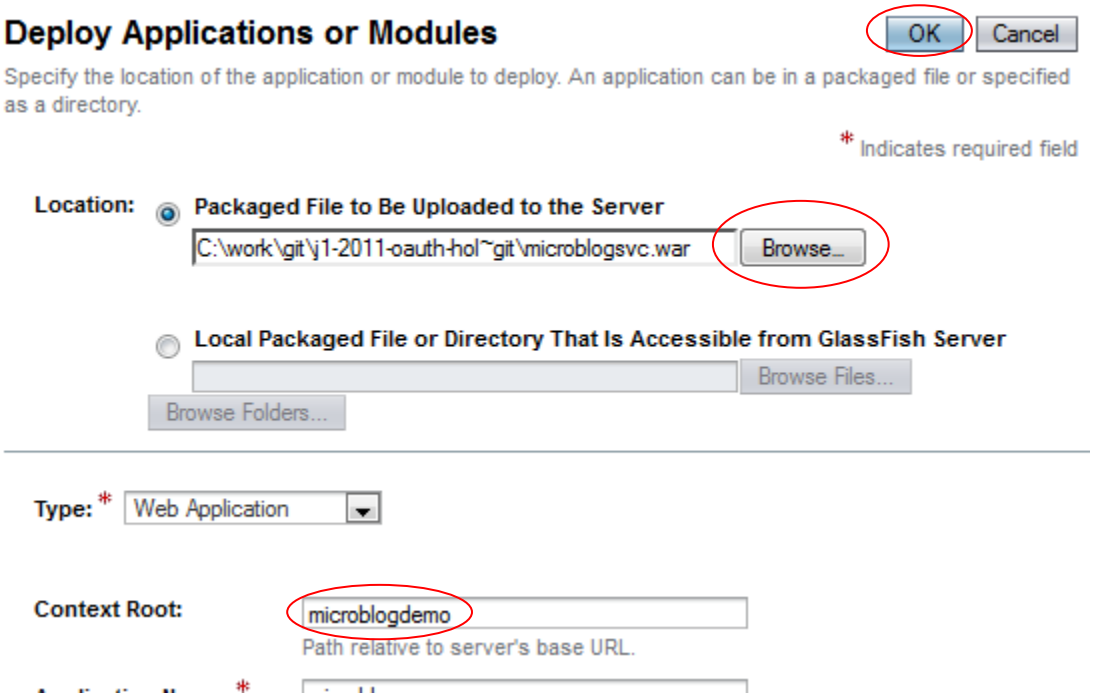
7. In the next screen click New, fill out the following info in the form that will show up and click OK:

   User ID: hol-user
   Group List: users
   New Password: javaone
   Confirm New Password: javaone

8. Now, click Applications on the left, then click Deploy.



9. In the next screen click Browse, then browse to the microblogsvc.war file located in the <lab_root> folder, click OK in the File Browse dialog, set Context Root to microblogdemo and click OK to deploy it.



10. Go back to NetBeans, right-click on the GlassFish server again and choose Restart from the pop-up menu.

- That concludes the setup.