**[Simple JAX-RS Web Service in Java with Spring and CXF](http://www.dreamsyssoft.com/blog/blog.php?/archives/7-Simple-JAX-RS-Web-Service-in-Java-with-Spring-and-CXF.html)**

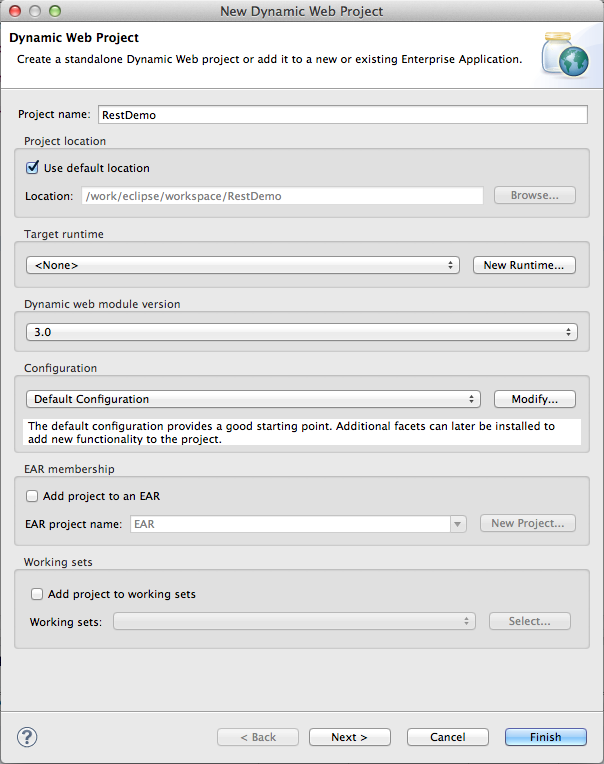
This is a simple tutorial on how to create a simple JAX-RS Web Service in Java using Spring and Apache CXF. This service will be follow the request/response pattern, it will using HTTP POSTs which are formatted JSON requests and it will produce JSON responses.

This will run as a simple war file on Tomcat 7. In the tutorial we will using Eclipse for development. This will show you how to create a JAX-RS Web Service for managing User objects, you can fetch, insert, update and delete them. It will use an in-memory store to keep it simple.

The full source code for this project is available [here on github](https://github.com/rockytriton/RestDemo/tree/master/RestDemo).

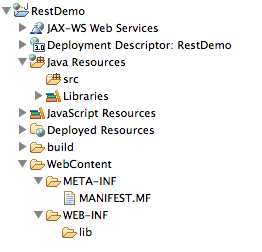
**Creating the Project**

The first step in this tutorial is to create a **Dynamic Web Project** in Eclipse. To do so, just choose File->New and select "Dynamic Web Project". If this option is not available, you will need to ensure that your version of eclipse has web development built in. I suggest starting with the "Eclipse for J2EE Developers" as a starting point.



Enter "RestDemo" as the project name and select all defaults, keep clicking next until you hit Finish.

Now you should have an empty Dynamic Web Project that looks like this:



Now we will move onto adding files to the project. I'm going to assume that you have some basic knowledge of how to add source files to Eclipse so that this tutorial doesn't get too verbose. The first thing that we need to add are the jar files that will be needed by this project. The best way to get these jar files is to download the source from github, here is a [link to the jar files](https://github.com/rockytriton/RestDemo/tree/master/RestDemo/WebContent/WEB-INF/lib). Copy each of these jar files into your project into the**/WebContent/WEB-INF/lib** folder.

**Create The Java Code**

Next we will start adding some Java source files. We are going to use **com.dreamsyssoft.demo.rest** as the base package for the service. We will start by adding our model objects, which will go in the package**com.dreamsyssoft.demo.rest.model**. We will create one main model object, **User** as well as a**UserRequest** and **UserResponse** class.

**User.java**

package com.dreamsyssoft.demo.rest.model;

import java.util.Date;

public class User

{

private Integer id;

private String name;

private String email;

private Date birthDate;

private String city;

private String state;

public Integer getId()

{

return id;

}

public void setId(Integer id)

{

this.id = id;

}

public String getName()

{

return name;

}

public void setName(String name)

{

this.name = name;

}

public String getEmail()

{

return email;

}

public void setEmail(String email)

{

this.email = email;

}

public Date getBirthDate()

{

return birthDate;

}

public void setBirthDate(Date birthDate)

{

this.birthDate = birthDate;

}

public String getCity()

{

return city;

}

public void setCity(String city)

{

this.city = city;

}

public String getState()

{

return state;

}

public void setState(String state)

{

this.state = state;

}

}

**UserRequest.java**

package com.dreamsyssoft.demo.rest.model;

public class UserRequest

{

private User user;

public User getUser()

{

return user;

}

public void setUser(User user)

{

this.user = user;

}

}

**UserResponse.java**

package com.dreamsyssoft.demo.rest.model;

import java.util.List;

public class UserResponse

{

private List users;

private String errorMessage;

private Boolean success = true;

public List getUsers()

{

return users;

}

public void setUsers(List users)

{

this.users = users;

}

public Boolean isSuccess()

{

return success;

}

public void setSuccess(Boolean success)

{

this.success = success;

}

public String getErrorMessage()

{

return errorMessage;

}

public void setErrorMessage(String errorMessage)

{

this.errorMessage = errorMessage;

}

}

The **User** class is a simple value object, it has a few properties to define a user in the system. The**UserRequest** class is also simple, all it holds is a User object. In the real world your Request object would contain more information such as a security context or some type of paging information, so it may seem a bit overkill here but I am including it to help show the pattern. The **UserResponse** class simply holds a list of returned User objects as well as a boolean success status and a String error message. These classes will be the input and output to your JAX-RS service.

Next we will create a simple DAO interface and implementation. It follows the interface-implementation pattern which we take advantage of with Spring so that we can (if necessary) dynamically plug-in different implementations.

**UserManagerDao.java**

package com.dreamsyssoft.demo.rest.dao;

import java.util.List;

import com.dreamsyssoft.demo.rest.model.User;

public interface UserManagerDao

{

public User fetchUserById(Integer id);

public List fetchAllUsers();

public void insertUser(User user);

public void updateUser(User user);

public void deleteUser(User user);

}

**UserManagerMemoryDao.java**

package com.dreamsyssoft.demo.rest.dao.impl;

import java.util.ArrayList;

import java.util.List;

import com.dreamsyssoft.demo.rest.dao.UserManagerDao;

import com.dreamsyssoft.demo.rest.model.User;

public class UserManagerMemoryDao implements UserManagerDao

{

private int nextUserId = 0;

List users = new ArrayList();

public User fetchUserById(Integer id)

{

for (User user : users)

{

if (user.getId() == id)

{

return user;

}

}

throw new RuntimeException("User Not Found: " + id);

}

public List fetchAllUsers()

{

return users;

}

public void insertUser(User user)

{

user.setId(nextUserId++);

users.add(user);

}

public void updateUser(User user)

{

User editUser = fetchUserById(user.getId());

editUser.setBirthDate(user.getBirthDate());

editUser.setCity(user.getCity());

editUser.setEmail(user.getEmail());

editUser.setName(user.getName());

editUser.setState(user.getState());

}

public void deleteUser(User user)

{

User delUser = fetchUserById(user.getId());

users.remove(delUser);

}

}

Again, this is a simple in-memory DAO that does not store the data in any persistent way, but if you decided to implement a database solution, you could simply plug-in the implementation with spring as will be shown below.

Now we will add the actual JAX-RS Web Service interface and implementation. The interface **UserManager**and implementation **UserManagerService** are shown below.

**UserManager.java**

package com.dreamsyssoft.demo.rest.services;

import javax.ws.rs.Consumes;

import javax.ws.rs.POST;

import javax.ws.rs.Path;

import javax.ws.rs.Produces;

import com.dreamsyssoft.demo.rest.model.UserRequest;

import com.dreamsyssoft.demo.rest.model.UserResponse;

@Consumes("application/json")

@Produces("application/json")

public interface UserManager

{

@POST

@Path("/fetchUserById/")

public UserResponse fetchUserById(UserRequest request);

@POST

@Path("/fetchAllUsers/")

public UserResponse fetchAllUsers(UserRequest request);

@POST

@Path("/insertUser/")

public UserResponse insertUser(UserRequest request);

@POST

@Path("/updateUser/")

public UserResponse updateUser(UserRequest request);

@POST

@Path("/deleteUser/")

public UserResponse deleteUser(UserRequest request);

}

**UserManagerService.java**

package com.dreamsyssoft.demo.rest.services.impl;

import java.util.Arrays;

import com.dreamsyssoft.demo.rest.dao.UserManagerDao;

import com.dreamsyssoft.demo.rest.model.UserRequest;

import com.dreamsyssoft.demo.rest.model.UserResponse;

import com.dreamsyssoft.demo.rest.services.UserManager;

public class UserManagerService implements UserManager

{

private UserManagerDao userDao;

public UserManagerDao getUserDao()

{

return userDao;

}

public void setUserDao(UserManagerDao userDao)

{

this.userDao = userDao;

}

public UserResponse fetchUserById(UserRequest request)

{

UserResponse response = new UserResponse();

try

{

response.setUsers(Arrays.asList(getUserDao().fetchUserById(

request.getUser().getId())));

}

catch (Exception e)

{

response.setSuccess(false);

response.setErrorMessage(e.getClass() + ": " + e.getMessage());

}

return response;

}

public UserResponse fetchAllUsers(UserRequest request)

{

UserResponse response = new UserResponse();

try

{

response.setUsers(getUserDao().fetchAllUsers());

}

catch (Exception e)

{

response.setSuccess(false);

response.setErrorMessage(e.getClass() + ": " + e.getMessage());

}

return response;

}

public UserResponse insertUser(UserRequest request)

{

UserResponse response = new UserResponse();

try

{

getUserDao().insertUser(request.getUser());

}

catch (Exception e)

{

response.setSuccess(false);

response.setErrorMessage(e.getClass() + ": " + e.getMessage());

}

return response;

}

public UserResponse updateUser(UserRequest request)

{

UserResponse response = new UserResponse();

try

{

getUserDao().updateUser(request.getUser());

}

catch (Exception e)

{

response.setSuccess(false);

response.setErrorMessage(e.getClass() + ": " + e.getMessage());

}

return response;

}

public UserResponse deleteUser(UserRequest request)

{

UserResponse response = new UserResponse();

try

{

getUserDao().deleteUser(request.getUser());

}

catch (Exception e)

{

response.setSuccess(false);

response.setErrorMessage(e.getClass() + ": " + e.getMessage());

}

return response;

}

}

The **UserManager** interface has several JAX-RS annotations. The first 2 you will see at the top level,**@Consumes** and **@Produces**, indicate that its requests consume and produce content-type**application/json**. Next on each method we give the annotation **@POST** which indicates that this method will be called on an HTTP POST only. Each method also has a **@Path** annotation which gives the URL path to this service method.

The implementation class **UserManagerService** is also very simple, it has a UserManagerDao object with a getter and setter and then a simple implementation of each of the interface methods. You will notice that we wrap each of these calls with a try/catch(Exception), this is so that we can always return a valid JSON response to the caller, even if they give bad data. So instead of returning an HTTP error along with some garbage HTML data response, we return an actual JSON UserResponse object with an error message set and the success flag set to false.

**Wiring it all together with Spring**

We've created all the code we need for this project, now we just need to wire it all together with the Spring Framework. You will need to create a folder named "conf" in /WebContent/WEB-INF. Inside the conf folder, create a new xml file named "rest-context.xml". This will be the spring context file.

**rest-context.xml**

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:cxf="http://cxf.apache.org/core"

xmlns:jaxws="http://cxf.apache.org/jaxws"

xmlns:jaxrs="http://cxf.apache.org/jaxrs"

xsi:schemaLocation="

http://cxf.apache.org/core http://cxf.apache.org/schemas/core.xsd

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://cxf.apache.org/jaxrs http://cxf.apache.org/schemas/jaxrs.xsd

http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

<import resource="classpath:META-INF/cxf/cxf.xml"/>

<import resource="classpath:META-INF/cxf/cxf-servlet.xml"/>

<cxf:bus>

<cxf:features>

<cxf:logging/>

</cxf:features>

</cxf:bus>

<bean id="userDao"

class="com.dreamsyssoft.demo.rest.dao.impl.UserManagerMemoryDao">

</bean>

<bean id="userManagerService"

class="com.dreamsyssoft.demo.rest.services.impl.UserManagerService">

<property name="userDao" ref="userDao"/>

</bean>

<bean id="jsonProvider"

class="org.codehaus.jackson.jaxrs.JacksonJsonProvider"/>

<jaxrs:server id="userManagerREST" address="/rest/UserManager">

<jaxrs:serviceBeans>

<ref bean="userManagerService"/>

</jaxrs:serviceBeans>

<jaxrs:providers>

<ref bean='jsonProvider' />

</jaxrs:providers>

</jaxrs:server>

</beans>

This is a typical Spring context file containing beans to be loaded by the Spring container. You can see where we create beans and give them an ID. For instance, we create an instance of our **UserManagerMemoryDao**class and give it the id **userDao**:

<bean id="userDao" class="com.dreamsyssoft.demo.rest.dao.impl.UserManagerMemoryDao">

</bean>

Then we create an instance of our **UserManagerService** implementation class. We set the "userDao" property of this class by giving it a reference to the bean we just created with the **userDao** bean ID. This is where things are injected by Spring. As long as you have a setter named setUserDao, it will derive that name from the property name "userDao" and call that method to inject the bean at runtime.

<bean id="userManagerService" class="com.dreamsyssoft.demo.rest.services.impl.UserManagerService">

<property name="userDao" ref="userDao"/>

</bean>

After this in the file we create a jsonProvider to give to the jaxrs:server element. In the jaxrs:server element, we give the address as an attribute, this will be the base location for the JAX-RS service. It will be**http://localhost:8080/RestDemo/services/rest/UserManager** based on this attribute given. Next we set the serviceBeans and add a reference to our userManagerService bean and then set the jaxrs:providers and give a reference to our jsonProvider bean.

Take some time to study over the XML and find the relationships between your code and the beans and the relationships between the beans themselves in the XML.

The only thing left to do now is to create a **web.xml** file to help us bundle this project into a WAR file. In the /WebContent/WEB-INF directory, create a file named **web.xml** with the given contents:

**web.xml**

<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee

http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"

metadata-complete="true">

<display-name>RestDemo</display-name>

<description>RestDemo</description>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/conf/\*-context.xml</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<servlet>

<servlet-name>CXFServlet</servlet-name>

<servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>CXFServlet</servlet-name>

<url-pattern>/services/\*</url-pattern>

</servlet-mapping>

</web-app>

This is a standard Java Web Application configuration file. There are several parts of this file which we care about specifically for this tutorial. First, we set a content-param to be the location of the spring context configuration file(s). In this example we use **/WEB-INF/conf/\*-context.xml**, you can use an exact filename but I prefer to split out my spring context files logically into separate files normally, so as long as they fit the pattern, they will be picked up by the **ContextLoaderListener** class which we add as a listener right below it:

**web.xml**

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/conf/\*-context.xml</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

That is essentially how you load the Spring container into your web application.

Finally we load the Apache CXF framework. We first create the **CXFServlet** servlet and then add the mapping for it to the URL pattern **/services/\***

**web.xml**

<servlet>

<servlet-name>CXFServlet</servlet-name>

<servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

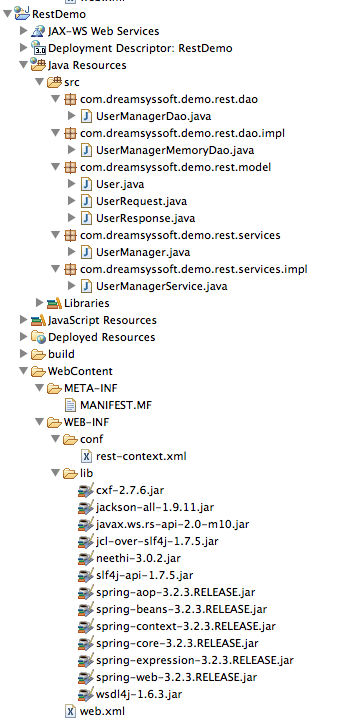
<servlet-name>CXFServlet</servlet-name>

<url-pattern>/services/\*</url-pattern>

</servlet-mapping>

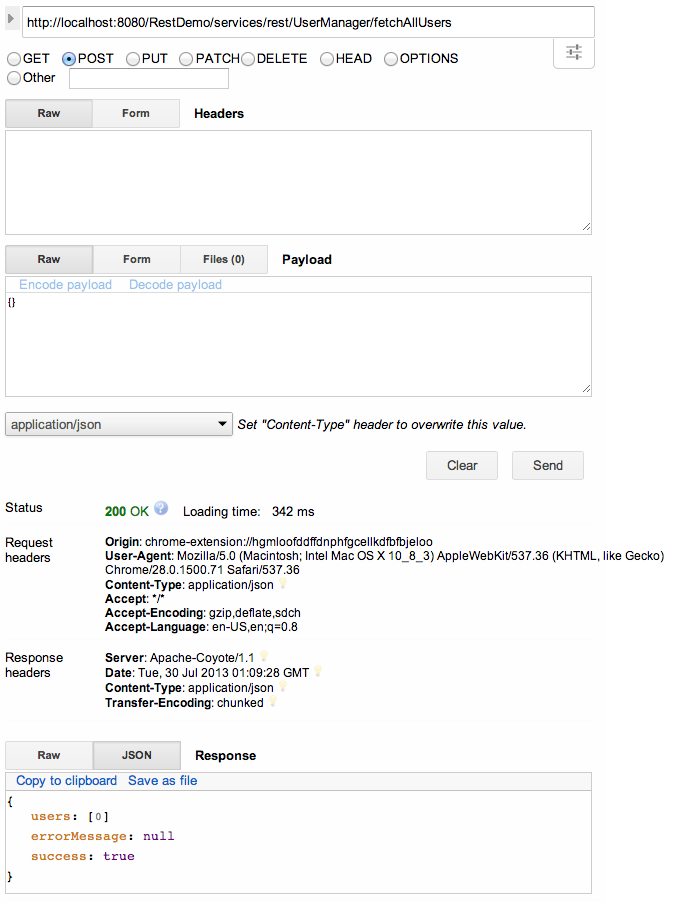
**Running the JAX-RS Web Service**

Now your service should be ready. Your project should look like the image below:



Now you can right-click on your **RestDemo** project and select "Export->WAR File". Select the destination to be your tomcat 7 webapps directory, then startup tomcat 7. I am going with the assumption that you already know how to install and run tomcat.

You are now ready to test your JAX-RS service, I would suggest using the [Advanced Rest Client Chrome Plug-in](https://chrome.google.com/webstore/detail/advanced-rest-client/hgmloofddffdnphfgcellkdfbfbjeloo?hl=en-US). It's an excellent free utility for testing REST services of all shapes and forms. Here is a screenshot of using it to test the very first fetchAllUsers method:



Notice that we set **http://localhost:8080/RestDemo/services/rest/UserManager/fetchAllUsers** as the URL for our fetchAllUsers request. For the payload, we simply put an empty JSON object **{}** and we can see the following JSON response:

{

users: [0]

errorMessage: null

success: true

}

Now let's create a user, set the URL to**http://localhost:8080/RestDemo/services/rest/UserManager/insertUser** and set the payload to:

{"user":{"name":"Rocky","email":"someting@nospam.org","birthDate":"1975-01-01","city":"Omaha","state":"NE"}}

This will give the response letting us know that the request was successful.

{

users: null

errorMessage: null

success: true

}

Now if we do our original fetchAllUsers request again, we will see this as the response:

{

users: [1]

0: {

id: 0

name: "Rocky"

email: "someting@nospam.org"

birthDate: 157766400000

city: "Omaha"

state: "NE"

}-

-

errorMessage: null

success: true

}

Have fun playing around with your new service, if you have any questions leave a comment, or use the "Contact Us" link to the left. Also, from the left side bar you can like this blog post on facebook or tweet it on twitter.