simeio

RESTful OIM

Exposing Oracle Identity Manager via REST APIs

By **Abhishek Gupta**, **Senior IAM Engineer** at Simeio Solutions

April 30, 2014

Introduction

Representational State Transfer, more commonly referred to as REST, is a simple, lightweight way to exchange data between web services or applications. REST owes its origin to Roy Thomas Fielding, an American computer scientist who in his doctoral dissertation (*Architectural Styles and the Design of Network-based Software Architectures*) described it as a critical architectural element of the World Wide Web.

REST has steadily evolved as the modern way of developing web service oriented applications. Key players such as *Google, eBay, Amazon, Twitter* and many others have introduced REST equivalents to their already existing stack of web services, which had previously relied only on Simple Object Access Protocol (SOAP) for data exchange.

REST and OIM

Oracle Identity Manager (OIM) is an enterprise identity management system that manages user's access privileges in enterprise IT resources by controlling users, roles, accounts and entitlements. The Oracle Identity Manager platform automates access rights management, security and provisioning of IT resources. It connects users to resources, and revokes and restricts unauthorized access to protect sensitive corporate information.

OIM also exposes its business functionality to external clients via two Application Programming Interfaces (APIs):

- **SOAP based** – Leverages *Service Provisioning Markup Language* (SPML) as the message protocol
- **Java based** – Exposes the *EJB* (business tier) layer of OIM in the form easy-to-use APIs

Can REST play a part?

With a RESTful interface for OIM, one can potentially have a fairly robust service layer, which can benefit from the some of the advantages that REST has to offer, including:

- A loosely coupled service architecture
- Cloud friendly (SaaS)
- Support for heterogeneous clients ranging from web-based (thin client) applications, GUI based (thick client applications) and Mobile Applications (Android, iOS, HTML5 based)
- Scalability
- Lightweight

Sound promising? Read on…

What This Article Covers

- A brief introduction to REST and the Java API for RESTful web services (JAX-RS)
- Walk through of a sample JEE application, which helps expose OIM information in a RESTful fashion

- An example of OIM APIs, representing the User entity within OIM as REST resources, and accessing them via a consistent interface

Basic knowledge of the following components is assumed:

- *REST* and its generic architectural principals
- *JAX-RS* – Java API for RESTful web services
- *JAXB* – Java Architecture for XML Binding
- *Oracle Identity Manager* APIs
- *WebLogic* Application Server

What This Article Does Not Intend to Do

- Provide detailed analysis about REST and its architectural principles
- Spark ***REST vs. SOAP*** debates.

Technologies/Components Used

- *Java API for RESTful web services* (JAX-RS) – For writing the core REST service implementation
- *Jersey* – Jersey is an open source reference implementation of JAX-RS
- *Java Architecture for XML Binding* (JAXB) – Standard APIs for Java object to XML binding
- *Oracle Identity Manager (OIM) 11g* (R2 PS1) installed on OEL 5.5 (64-bit)
- *WebLogic 10.3.6* – The Application Server hosting OIM as well as the RESTful web service application
- *Java Development Kit* (JDK) 1.7

A Brief Introduction to REST

REST is an architectural style that closely mimics the operational semantics of the web. REST is *not* a framework or API, but it offers a simple way to handle create-update-and-delete (CRUD) operations on data using APIs or frameworks over the Internet.
There are several key features of REST based services:

- **They rely on *HTTP* as the transfer protocol and follow the request-response model** – HTTP is a ubiquitous protocol. It empowers service usability without any additional requirements beyond being able to exchange the data formats the service is expecting.

- **They use HTTP verbs (*GET, PUT, POST, DELETE,* etc.) for executing operations** – The web consists of well-identified resources linked together and accessed through simple HTTP requests. The main types of requests standardized in HTTP are GET, POST, PUT and DELETE. These are also called verbs, commands or methods. HTTP defines four other methods that are less frequently used: HEAD, TRACE, OPTIONS and CONNECT.

- **They are inherently stateless** – HTTP operations/requests should be executed in isolation and should not take into consideration previous invocations or store any resource specific information (state). It does not mean that RESTful applications can't have state. Stateless implies that there is *no client session data stored on the server*. Any session-specific data should be held and maintained by the client and transferred to the server with each request.

- **They are based around *Resources* and their *Representations*** – Representations are exchanged between the client and service. These representations could be XML, JSON or any custom format.
- **They have addressability, i.e. they expose a *consistent* interface (*URI*) for accessing data/resources** – Representations can be accessed/acted upon through meaningful URIs, which are transmitted via HTTP requests.
- **They embrace *Hypermedia As The Engine Of Application State* (HATEOAS)** – Hypermedia is a document-centric approach with the added support for embedding links to other services and information within that document format. One of the uses of hypermedia and hyperlinks is composing complex sets of information from disparate sources.

Java API for RESTful Web Services (JAX-RS)

Java API for RESTful Web Services (JAX-RS) is a standard API/framework for writing REST based web services using Java.

There are several salient features of JAX-RS:

- **POJO-based** – JAX-RS provides a set of annotations and associated classes/interfaces that may be used with POJOs in order to expose them as Web resources.
- **HTTP-centric –** JAX-RS provides a clear mapping between HTTP and URI elements and the corresponding API classes and annotations.
- **Format independence** – The API is applicable to a wide variety of HTTP entity body content types. It provides the necessary extensibility to allow additional types to be added by an application in a standard manner.

JAX-RS offers several benefits, including:

- The Annotation driven API is robust, extensible and easy-to-use
- It shields developers from low-level HTTP, XML and Servlet related plumbing code
- Developers can choose from a variety of implementations from different vendors (In this demo, we will be using *Jersey*)

OIM on REST

Armed with this brief introduction, lets now demonstrate how one can leverage JAX-RS in order to develop an application which can expose user information within OIM over a RESTful interface. This will involve:

- Writing the business logic in the form of a JAX-RS application
- Packaging and deploying the application on WebLogic 10.3.6
- Testing the service

Writing the JAX-RS Application/Service

The high level steps here are to:

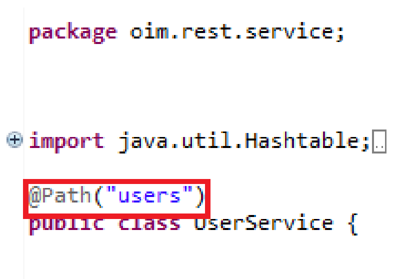
1. Write the core service class
2. Write the Domain/Entity classes
3. Code the Entity Transformer classes

Step 1: Write the Core Service/Resource class

In JAX-RS terminology, a “service” is nothing but a Java class that:

- Is decorated with JAX-RS annotations
- Binds/maps incoming HTTP requests to appropriate Java methods that can “service” these requests.

UserService.java **is the service class being used in this example**

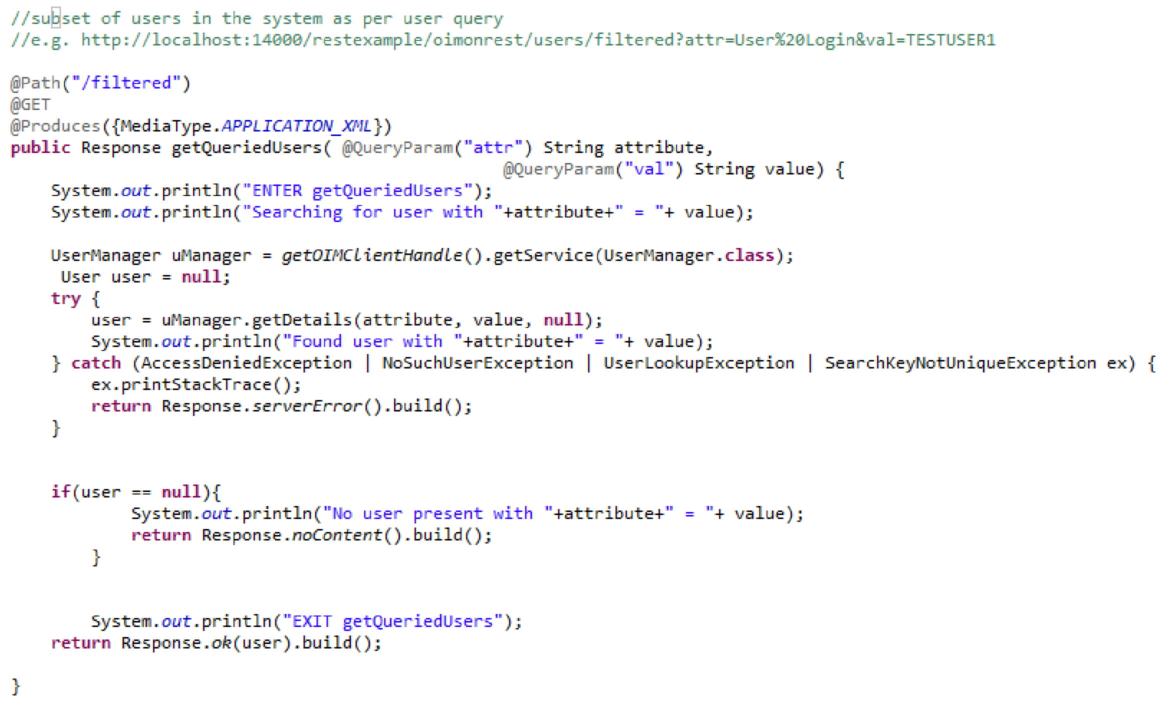


```
package oim.rest.service;


import java.util.Hashtable;

@Path("users")
public class UserService {
```

Let’s start with some sample code designed to search for ALL the users within OIM:

```
//Fetch all users in the system

//e.g. http://localhost:14000/restexample/oimonrest/users/all

@Path("/all")
@GET
@Produces({MediaType.APPLICATION_XML})
@MarkerAnnotation

public Response getAllUsersInOIM() {
    System.out.println("ENTER getAllUsersInOIM");

    UserManager uManager = getOIMClientHandle().getService(UserManager.class);
     List<User> users = null;
    SearchCriteria criteria = new SearchCriteria("User Login", "*", SearchCriteria.Operator.EQUAL);
    try {
       users = uManager.search(criteria, null, null);
    } catch (AccessDeniedException | UserSearchException ex) {
        ex.printStackTrace();
        return Response.serverError().build();
    }

    if(users.size() < 1){
            System.out.println("No users present in the system");
            return Response.noContent().build();
        }
        System.out.println("Number of users provisioned in the system  = "+ users.size());

        System.out.println("EXIT getAllUsersInOIM");
   return Response.ok(users).build();

}
```

Here’s another example of searching for a specific user via a filtration or search criterion:



```
//subset of users in the system as per user query
//e.g. http://localhost:14000/restexample/oimonrest/users/filtered?attr=User%20Login&val=TESTUSER1

@Path("/filtered")
@GET
@Produces({MediaType.APPLICATION_XML})
public Response getQueriedUsers( @QueryParam("attr") String attribute,
                                                 @QueryParam("val") String value) {
    System.out.println("ENTER getQueriedUsers");
    System.out.println("Searching for user with "+attribute+" = "+ value);

    UserManager uManager = getOIMClientHandle().getService(UserManager.class);
     User user = null;
    try {
        user = uManager.getDetails(attribute, value, null);
        System.out.println("Found user with "+attribute+" = "+ value);
    } catch (AccessDeniedException | NoSuchUserException | UserLookupException | SearchKeyNotUniqueException ex) {
        ex.printStackTrace();
        return Response.serverError().build();
    }


    if(user == null){
            System.out.println("No user present with "+attribute+" = "+ value);
            return Response.noContent().build();
        }


        System.out.println("EXIT getQueriedUsers");
    return Response.ok(user).build();

}
```

Now let’s examine each of the key elements in these code samples:

- **@javax.ws.rs.Path annotation** – This Identifies the URI path that a resource class or class method will serve requests for. In this sample, this annotation has been used to decorate both the resource class, as well as the service methods. The final URL is a combination of the values of the annotations on the class and the individual methods. In our example, this resource class exposes two capabilities/services.

  How are these capabilities accessed by external clients/consumers? JAX-RS runtime binds incoming HTTP requests to appropriate service methods that contain the business logic. This is enabled via the *@javax.ws.rs.GET* annotation.

- **@javax.ws.rs.GET annotation** – This indicates that the annotated method responds to HTTP GET requests.

  How can the service communicate the message protocol and type to the prospective consumer? For that, we turn to the *@javax.ws.rs.Produces* annotation.

- **@javax.ws.rs.Produces annotation** – This defines the media type(s) that the methods of a resource class can produce (directly or via a *javax.ws.rs.ext.MessageBodyWriter*). In this sample, this annotation explicitly states that the methods will return an *XML payload* (*refer to screenshots below*)

The User information within OIM is returned in an XML format (single user or a group of users).

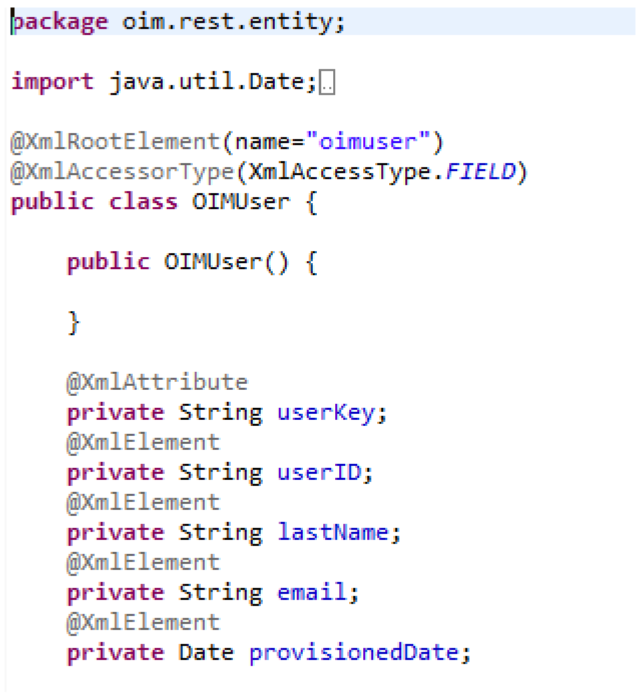
Step 2: Write the Domain/Entity classes

The entities generally represent the domain model of a system, i.e. the object that encapsulates and propagates business data within the system.

Our domain classes are JAXB annotated classes, which encapsulate the attributes associated with a user along with accessor/mutator methods. They can be extended and enhanced as required. Only a limited set of user attributes have been shown within this example.

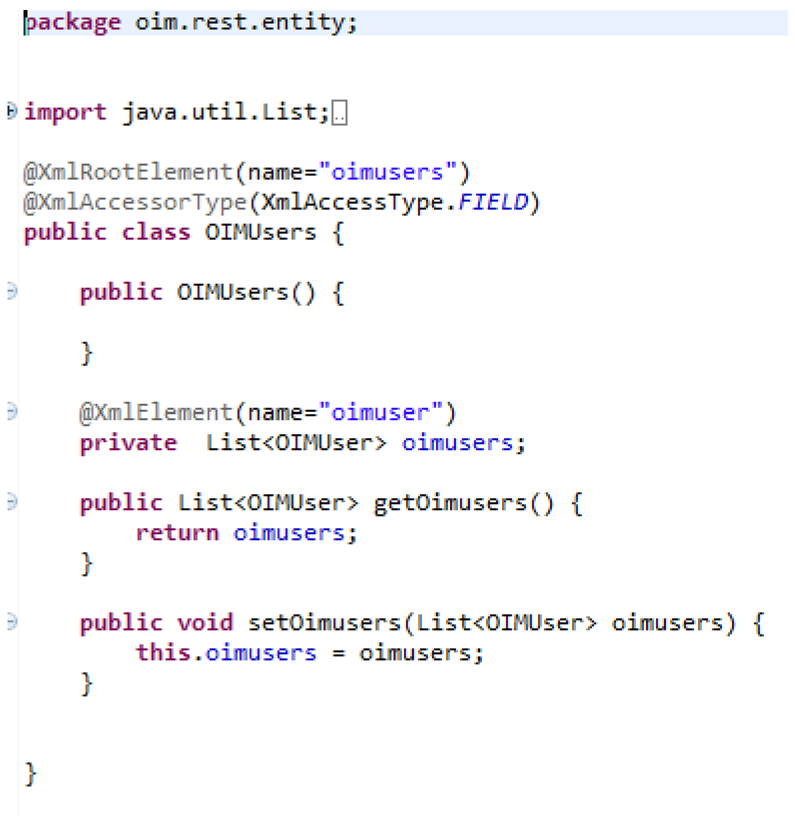
Note: As mentioned earlier, these classes are standard JAXB annotated classes. Explanation of the JAXB annotations and the Java/XML binding concepts are out of scope of this article. For more information, see the References section at the end of this article, which provides a link to resources from where you can familiarize yourself with JAXB.

OIMUser.java - represents the information of a single user



```
package oim.rest.entity;

import java.util.Date;

@XmlRootElement(name="oimuser")
@XmlAccessorType(XmlAccessType.FIELD)
public class OIMUser {

    public OIMUser() {

    }

    @XmlAttribute
    private String userKey;
    @XmlElement
    private String userID;
    @XmlElement
    private String lastName;
    @XmlElement
    private String email;
    @XmlElement
    private Date provisionedDate;
```

OIMUsers.java – A JAXB annotated class representing a list of users



```
package oim.rest.entity;


import java.util.List;

@XmlRootElement(name="oimusers")
@XmlAccessorType(XmlAccessType.FIELD)
public class OIMUsers {

    public OIMUsers() {

    }

    @XmlElement(name="oimuser")
    private  List<OIMUser> oimusers;

    public List<OIMUser> getOimusers() {
        return oimusers;
    }

    public void setOimusers(List<OIMUser> oimusers) {
        this.oimusers = oimusers;
    }


}
```

Step 3: Code the Entity Transformer Classes

You might be asking, "Why do we need custom JAXB annotated domain classes?" It's because OIM API already provides an *oracle.iam.identity.usermgmt.vo.User* class which represents a user within OIM.

Why, then, can't we just return an instance of oracle.iam.identity.usermgmt.vo.User via our RESTful service? If we directly return instances of *oracle.iam.identity.usermgmt.vo.User* class form our service, we'll get this error:

> *<Error> <com.sun.jersey.spi.container.ContainerResponse> <BEA-000000> <A message body writer for Java class oracle.iam.identity.usermgmt.vo.User, and Java type class oracle.iam.identity.usermgmt.vo.User, and MIME media type application/xml was not found>*

It's evident that the *oracle.iam.identity.usermgmt.vo.User class is not compatible with JAXB.* ***Hence it is not possible to marshal/serialize an instance of the oracle.iam.identity.usermgmt.vo.User class to its XML format directly*.**

The solution is simple and elegant and the best part is that it is available OOTB in JAX-RS. Custom serialization or payload manipulation/transformation is with JAX-RS is powered by *Entity Providers.*

Entity Transformers/Providers

These components are instrumental in housing the logic that transforms the incompatible *oracle.iam.identity.usermgmt.vo.User* object returned via OIM into our custom domain class.

Provider classes implement *javax.ws.rs.ext.MessageBodyWriter* interface and provide the transformation logic from the *oracle.iam.identity.usermgmt.vo.User* to our custom entity class (OIMUser.java, OIMUsers.java) within the *writeTo* method.

The JAXB compatible entity representation can then be seamlessly marshaled/serialized by JAX-RS runtime into its XML representation.

The javax.ws.rs.ext.MessageBodyWriter Interface

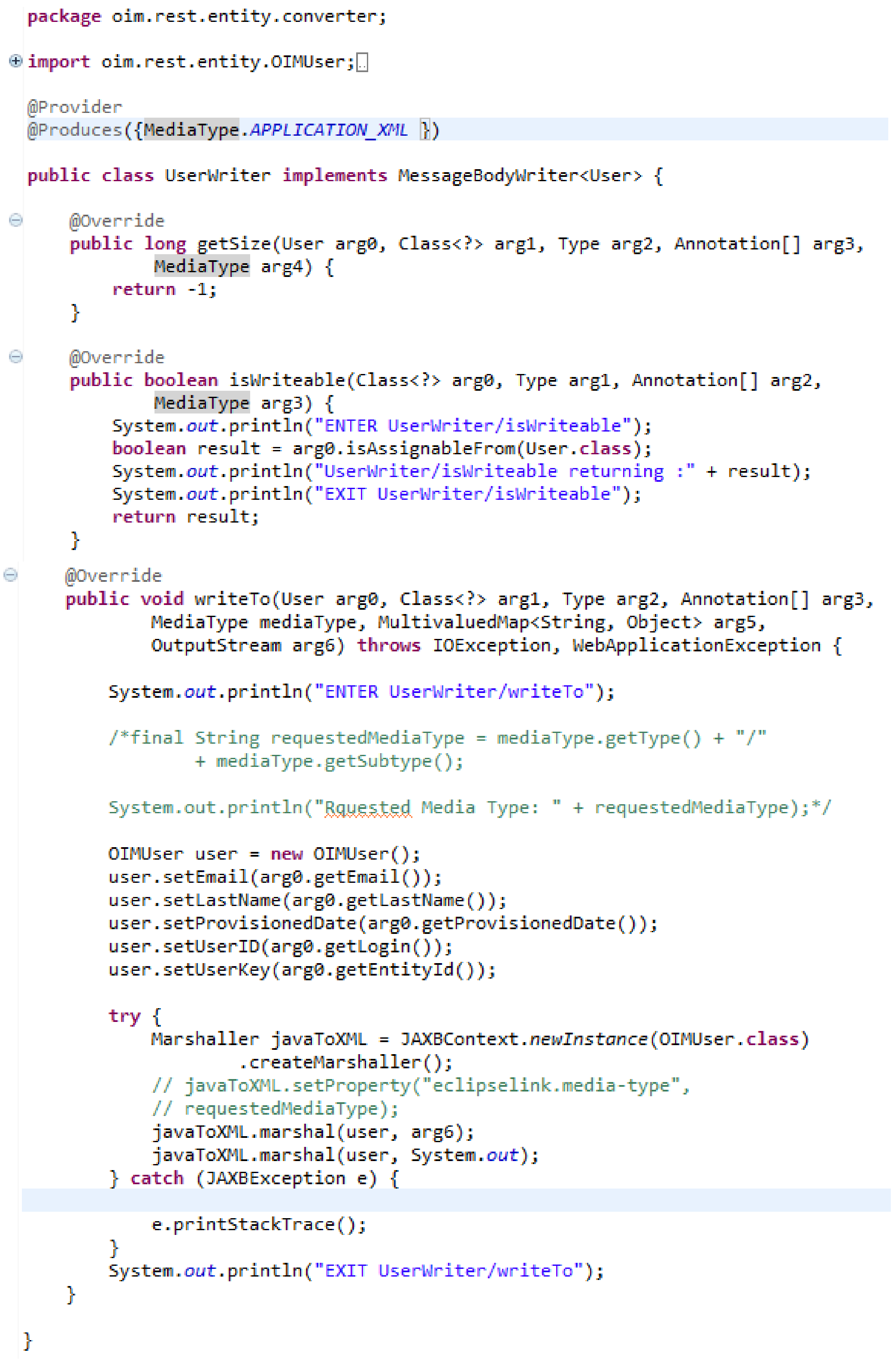
The javax.ws.rs.ext.MessageBodyWriter interface acts as a contract for a provider that supports the conversion of a Java type to a stream. In this example, we are leveraging this interface to convert the *oracle.iam.identity.usermgmt.vo.User* into an instance of the *OIMUser.java* (refer ***Domain/Entity classes*** section), which in turn is automatically serialized/de-serialized by the JAXB runtime.

Providers implementing this interface must be either programmatically registered in a JAX-RS runtime or must be annotated with @Provider annotation to be automatically discovered by the JAX-RS runtime during a provider-scanning phase.

The methods of the MessageBodyWriter interface are:

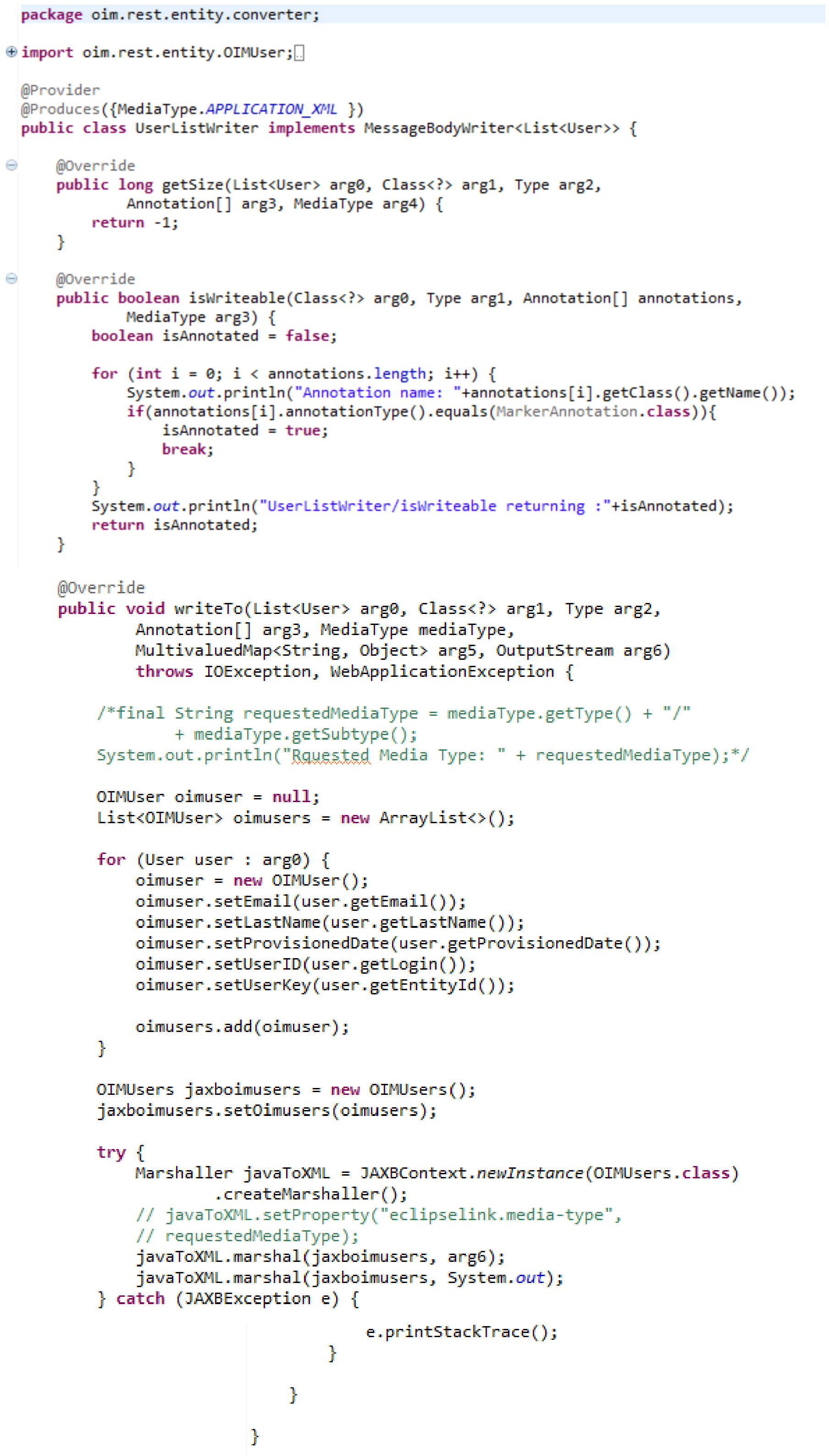
- ***isWriteable*** – Determines if the implementation supports a particular type. In this sample, the *oracle.iam.identity.usermgmt.vo.User* class is the type that is being used to determine the outcome of the method. This is the first method that is invoked by the JAX-RS runtime.
- ***getSize*** – Calculates the length of the serialized message. It is represented in the form of the HTTP Content-Length header. **Note:** This is deprecated in JAX-RS 2.0. Implementations are advised to return -1 as a default value.
- ***writeTo*** – *Contains* the logi**c t**o write a type (class) to an HTTP message in a serialized form. This is the bread and butter method of this interface and contains the primary business logic pertaining to the serialization/transformation of the message payload from one form to the final format that the consumer intends to receive.

UserWriter.java - A JAX-RS entity provider for handling a single instance of oracle.iam.identity.usermgmt.vo.User



```
package oim.rest.entity.converter;

import oim.rest.entity.OIMUser;

@Provider
@Produces({MediaType.APPLICATION_XML })

public class UserWriter implements MessageBodyWriter<User> {

    @Override
    public long getSize(User arg0, Class<?> arg1, Type arg2, Annotation[] arg3,
            MediaType arg4) {
        return -1;
    }

    @Override
    public boolean isWriteable(Class<?> arg0, Type arg1, Annotation[] arg2,
            MediaType arg3) {
        System.out.println("ENTER UserWriter/isWriteable");
        boolean result = arg0.isAssignableFrom(User.class);
        System.out.println("UserWriter/isWriteable returning :" + result);
        System.out.println("EXIT UserWriter/isWriteable");
        return result;
    }

    @Override
    public void writeTo(User arg0, Class<?> arg1, Type arg2, Annotation[] arg3,
            MediaType mediaType, MultivaluedMap<String, Object> arg5,
            OutputStream arg6) throws IOException, WebApplicationException {

        System.out.println("ENTER UserWriter/writeTo");

        /*final String requestedMediaType = mediaType.getType() + "/"
                + mediaType.getSubtype();

        System.out.println("Rquested Media Type: " + requestedMediaType);*/

        OIMUser user = new OIMUser();
        user.setEmail(arg0.getEmail());
        user.setLastName(arg0.getLastName());
        user.setProvisionedDate(arg0.getProvisionedDate());
        user.setUserID(arg0.getLogin());
        user.setUserKey(arg0.getEntityId());

        try {
            Marshaller javaToXML = JAXBContext.newInstance(OIMUser.class)
                    .createMarshaller();
            // javaToXML.setProperty("eclipselink.media-type",
            // requestedMediaType);
            javaToXML.marshal(user, arg6);
            javaToXML.marshal(user, System.out);
        } catch (JAXBException e) {

            e.printStackTrace();
        }
        System.out.println("EXIT UserWriter/writeTo");
    }

}
```

UserListWriter.java - A JAX-RS entity provider for handling a list of instances of oracle.iam.identity.usermgmt.vo.User



```java
package oim.rest.entity.converter;

import oim.rest.entity.OIMUser;

@Provider
@Produces({MediaType.APPLICATION_XML })
public class UserListWriter implements MessageBodyWriter<List<User>> {

    @Override
    public long getSize(List<User> arg0, Class<?> arg1, Type arg2,
            Annotation[] arg3, MediaType arg4) {
        return -1;
    }

    @Override
    public boolean isWriteable(Class<?> arg0, Type arg1, Annotation[] annotations,
            MediaType arg3) {
        boolean isAnnotated = false;

        for (int i = 0; i < annotations.length; i++) {
            System.out.println("Annotation name: "+annotations[i].getClass().getName());
            if(annotations[i].annotationType().equals(MarkerAnnotation.class)){
                isAnnotated = true;
                break;
            }
        }
        System.out.println("UserListWriter/isWriteable returning :"+isAnnotated);
        return isAnnotated;
    }

    @Override
    public void writeTo(List<User> arg0, Class<?> arg1, Type arg2,
            Annotation[] arg3, MediaType mediaType,
            MultivaluedMap<String, Object> arg5, OutputStream arg6)
            throws IOException, WebApplicationException {

        /*final String requestedMediaType = mediaType.getType() + "/"
                + mediaType.getSubtype();
        System.out.println("Rquested Media Type: " + requestedMediaType);*/

        OIMUser oimuser = null;
        List<OIMUser> oimusers = new ArrayList<>();

        for (User user : arg0) {
            oimuser = new OIMUser();
            oimuser.setEmail(user.getEmail());
            oimuser.setLastName(user.getLastName());
            oimuser.setProvisionedDate(user.getProvisionedDate());
            oimuser.setUserID(user.getLogin());
            oimuser.setUserKey(user.getEntityId());

            oimusers.add(oimuser);
        }

        OIMUsers jaxboimusers = new OIMUsers();
        jaxboimusers.setOimusers(oimusers);

        try {
            Marshaller javaToXML = JAXBContext.newInstance(OIMUsers.class)
                    .createMarshaller();
            // javaToXML.setProperty("eclipselink.media-type",
            // requestedMediaType);
            javaToXML.marshal(jaxboimusers, arg6);
            javaToXML.marshal(jaxboimusers, System.out);
        } catch (JAXBException e) {
                        e.printStackTrace();
                    }

                }

            }
```

A Note About javax.ws.rs.ext.MessageBodyReader

This interface is complimentary to the *MessageBodyWriter* interface – that is, it represents a contract for a provider/implementation that supports the conversion of a stream to a Java type.

- It is useful when the input messages to the REST service need to be represented as standard Java objects (JAXB entities or custom objects)
- *MessageBodyReader* implementations can be plugged in for the JAX-RS runtime to be able to seamlessly convert the consumer messages into Java object instances.

Detailed discussion about this interface is out of scope. Please refer to the References section at the end of this article for more on this.

Packaging and Deploying the Application in WebLogic 10.3.6

In this example, we are using WebLogic 10.3.6 as our application server.

The RESTful service should be packaged and deployed on the same server/domain where OIM has been installed.

Packaging

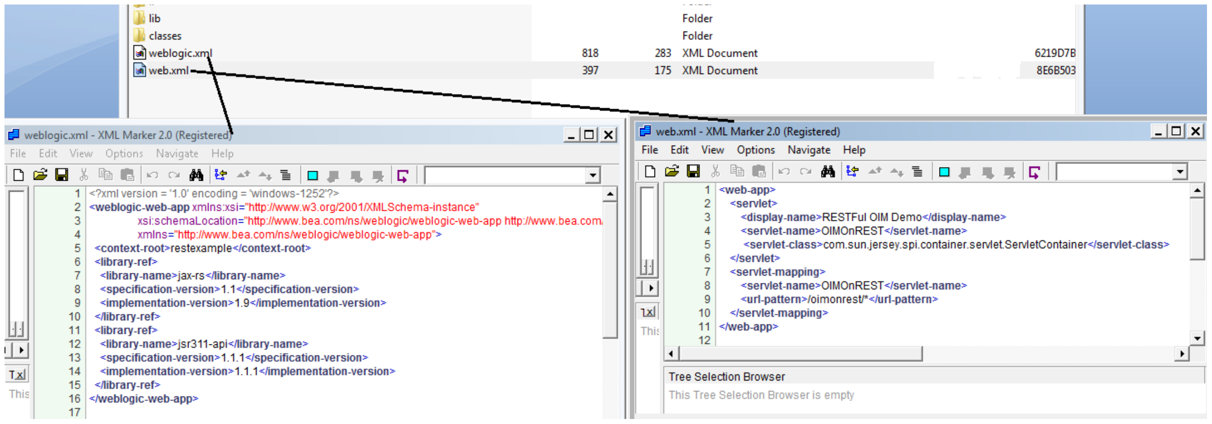
As mentioned earlier, we shall be using Jersey as the runtime JAX-RS provider. WebLogic 10.3.6 version needs to be configured in order to get things working. I will follow the *DRY (Don't Repeat Yourself)* principle and provide a link to the official WebLogic documentation, which talks about the same:

http://docs.oracle.com/cd/E23943_01/web.1111/e13734/rest.htm#CHDCGFCH

Essentially, the high level steps are as follows:

- Register Jersey JAX-RS RI Shared Libraries
- Configure your web application to user Jersey JAX-RS RI - add entries to the *web.xml and weblogic.xml*

This is how the packaged WAR file should look:


lib
classes
weblogic.xml
web.xml
Folder
Folder
818 283 XML Document 6219D7B
397 175 XML Document 8E6B503
weblogic.xml - XML Marker 2.0 (Registered)
web.xml - XML Marker 2.0 (Registered)
File Edit View Options Navigate Help
Tree Selection Browser
This Tree Selection Browser is empty

```
<?xml version = '1.0' encoding = 'windows-1252'?>
<weblogic-web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://www.bea.com/ns/weblogic/weblogic-web-app http://www.bea.com
            xmlns="http://www.bea.com/ns/weblogic/weblogic-web-app">
  <context-root>restexample</context-root>
  <library-ref>
    <library-name>jax-rs</library-name>
    <specification-version>1.1</specification-version>
    <implementation-version>1.9</implementation-version>
  </library-ref>
  <library-ref>
    <library-name>jsr311-api</library-name>
    <specification-version>1.1.1</specification-version>
    <implementation-version>1.1.1</implementation-version>
  </library-ref>
</weblogic-web-app>
```

```
<web-app>
  <servlet>
    <display-name>RESTFul OIM Demo</display-name>
    <servlet-name>OIMOnREST</servlet-name>
     <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>OIMOnREST</servlet-name>
    <url-pattern>/oimonrest/*</url-pattern>
  </servlet-mapping>
</web-app>
```

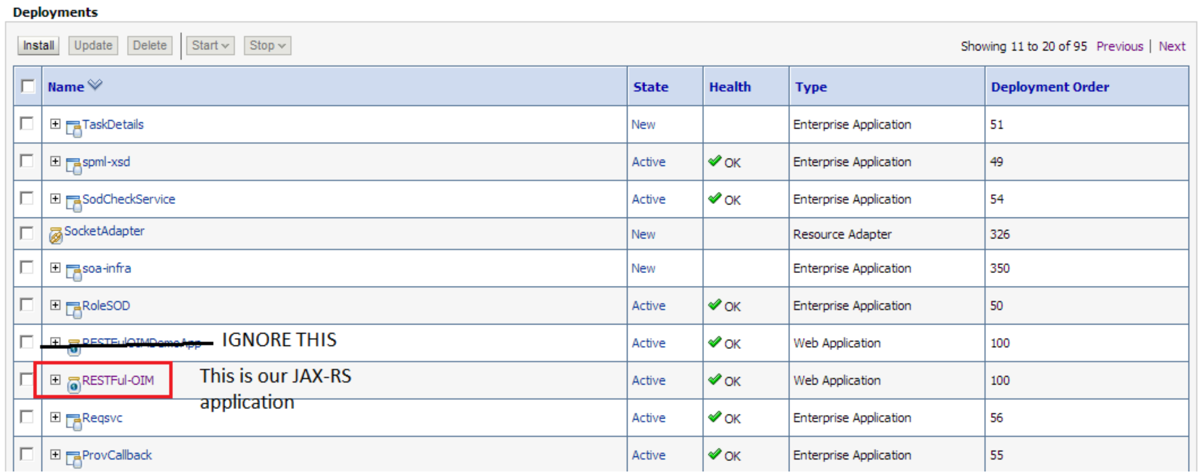
- **lib directory** – contains the helper classes/JAR files. In our case, it is the ***oimclient.jar***
- **classes directory** – contains the JAX-RS application classes (explained above). This is the business logic of our application
- **web.xml and weblogic.xml** – standard JavaEE deployment descriptors

Deployment

The deployment procedure is straightforward:

- The application should be deployed just like any other WAR file – all the default options can be accepted
- *Make sure that the WAR is deployed on the same server where OIM is installed*

Post Deployment



Name	State	Health	Type	Deployment Order
TaskDetails	New		Enterprise Application	51
spml-xsd	Active	✓ OK	Enterprise Application	49
SodCheckService	Active	✓ OK	Enterprise Application	54
SocketAdapter	New		Resource Adapter	326
soa-infra	New		Enterprise Application	350
RoleSOD	Active	✓ OK	Enterprise Application	50
~~RESTFulOIMDemoApp~~ IGNORE THIS	Active	✓ OK	Web Application	100
RESTFul-OIM This is our JAX-RS application	Active	✓ OK	Web Application	100
Reqsvc	Active	✓ OK	Enterprise Application	56
ProvCallback	Active	✓ OK	Enterprise Application	55

Testing/Accessing the Application

Case 1: Search ALL Users in OIM

To test this, browse to the following URL:

http://YOUR-SERVER-IP:PORT/restexample/oimonrest/users/all

e.g. http://192.168.0.XXX:14000/restexample/oimonrest/users/all

Expected Result:

```
<oimusers>
  <oimuser userKey="1">
    <userID>XELSYSADM</userID>
    <lastName>Administrator</lastName>
    <email>donotreply@oracle.com</email>
  </oimuser>
```

```
            <oimuser userKey="21">
               <userID>TESTUSER1</userID>
               <lastName>testuser1</lastName>
               <email>testuser1@email.com</email>
               <provisionedDate>2013-11-13T13:18:26.0</provisionedDate>
            </oimuser>
            <oimuser userKey="61">
               <userID>TESTUSER2</userID>
               <lastName>testuser2</lastName>
               <email>testuser2@email.com</email>
               <provisionedDate>2013-11-18T11:01:12.0</provisionedDate>
            </oimuser>
            <oimuser userKey="62">
               <userID>TESTUSER3</userID>
               <lastName>testuser3</lastName>
               <email>testuser3@email.com</email>
               <provisionedDate>2013-12-20T12:43:41.0</provisionedDate>
 </oimusers>
```

Case 2: Search For a Specific User (Filtration)

To test this, browse to the following URL:

http://YOUR-SERVER-IP:PORT/ restexample/oimonrest/users/filtered?attr=User Login&val=<USERID OF THE USER>

e.g. http://192.168.0.XXX:14000/restexample/oimonrest/users/filtered?attr=User Login&val=TESTUSER2

Expected Result:

```
      <oimuser userKey="21">
         <userID>TESTUSER1</userID>
         <lastName>testuser1</lastName>
         <email>testuser1@email.com</email>
         <provisionedDate>2013-11-13T13:18:26.0</provisionedDate>
      </oimuser>
```

You can also use other attributes apart from ***User Login*** as the search criteria e.g. *Email, First Name, Last Name* etc. Please note that ***User Login, Email, First Name, Last Name are 'keywords' within Oracle Identity Manager*** itself.

Note: The application URL depends upon the configuration within the web.xml and weblogic.xml:

- Context Root (*restexample*) comes from the *web.xml* and
- REST service URL (*oimonrest*) is configured within the *weblogic.xml* descriptor

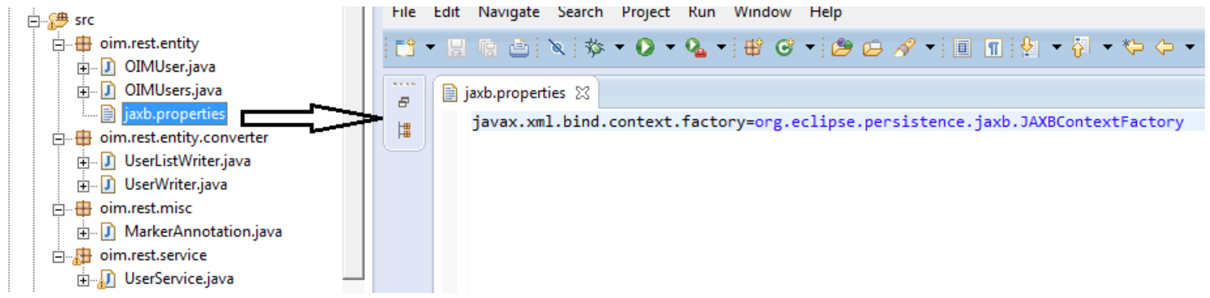
Further Enhancements

Adding JavaScript Object Notation (JSON) Support

Our sample application is now capable of returning user information within OIM as an XML payload to the client. We can further enrich the service by adding support for JavaScript Object Notation (JSON) as the messaging protocol between the client and the server.

I will not delve into the details in this article. I will, however, provide an overview of how this can be achieved and leave this as your homework assignment :-)

- **Leveraging EclipseLink Moxy** – Eclipse link MOxy provides a valuable feature, via which JAXB annotated classes can be directly serialized into JSON representations without ANY significant coding/programming efforts!
- **Deploying EclipseLink as a Shared Library** – In order to leverage the JSON binding feature, one would need to register EclipseLink as a shared library within Weblogic. More on this can be found here: http://docs.oracle.com/cd/E23943_01/web.1111/e13702/deploy.htm#i1021579
- **Configuring our application to use EclipseLink as the JAXB provider** – Using *jaxb.properties* within our entity packages



```
javax.xml.bind.context.factory=org.eclipse.persistence.jaxb.JAXBContextFactory
```

- **Specifying the media type** – Indicate the desired media type to our ***javax.xml.bind.Marshaller*** object during serialization:

```
javaToXML.setProperty("eclipselink.media-type", "application/json");
```

Conclusion

REST and RESTful web services have well and truly become the cornerstone for organizations for building loosely coupled, scalable, efficient and distributed systems. It has brought the "web" back into web services.

Exposing information within OIM via RESTful interfaces gives us several advantages ranging from lightweight implementation to versatile support for a number of consumers (Web or Mobile).

Questions?

Please visit us at http://www.simeiosolutions.com/

I hope this article and the example within can help you get started in building your own RESTful implementations.

Note: The sample application did not pay special attention to exception handling and security (authentication/authorization). These are definitely things which one should be looking into while trying to build a RESTful gateway for OIM.

References

Please leverage the references below for further reading and guidelines:

Oracle Identity Manager API reference: http://docs.oracle.com/cd/E37115_01/apirefs.1112/e28159/toc.htm

Oracle Identity Manager: http://www.oracle.com/technetwork/middleware/id-mgmt/overview/index-098451.html

REST principles (Roy Thomas Fielding's doctoral dissertation): http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Jersey: https://jersey.java.net/

JAX-RS: https://jax-rs-spec.java.net/

JAXB: https://jaxb.java.net/

Eclipse Link MoXY: http://www.eclipse.org/eclipselink/moxy.php

JSON: http://www.json.org/

Hypermedia as the Engine of Application State (HATEOAS): http://en.wikipedia.org/wiki/HATEOAS

SIMEIO SOLUTIONS
Hoboken Business Center
50 Harrison Street, Suite 304
Hoboken, NJ 07030
Phone: +1-201-239-1700