Component-Based Software Development

RESTful Web Services

Dr. Vinod Dubey
SWE 645
George Mason University

Acknowledgement

Content in this presentation has been adapted from chapter 31of Java How to program, 9th Edition by P.J. Deitel and H.M. Deitel; and from Java EE7 Essentials by Arun Gupta

OBJECTIVES

- Web Service Basics
- REST and RESTful Web Services – key characteristics
- JAX-RS Spec
- How to create RESTful services using Jersey
  - web.xml
- Client API
- Using a JPA Entity as a RESTful resource

Web Service Basics

- Web services **provide a mechanism to execute business operations remotely using standard based technologies and protocols such as XML/JSON and HTTP**

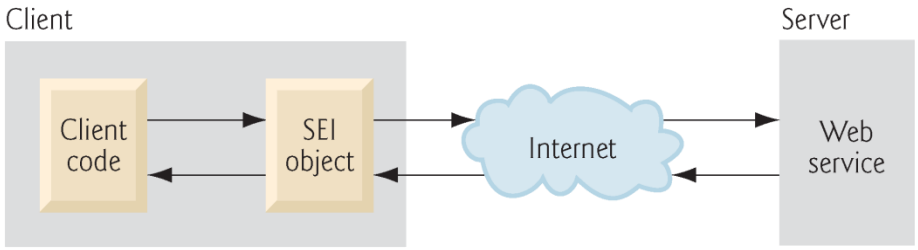


Fig. 31.7 | Interaction between a web service client and a web service.

Web Service Basics

- **W3C Definition of Web Service:** "A **software system** designed to support **interoperable machine-to-machine interaction** over a network."

- **Web services are platform and language independent**
  - Enabled by leveraging HTTP and JSON/XML, which are widely supported by libraries across every programming languages

Two Prevailing Styles of Web Services

- **RESTful Web Services**
  - Web services that are **based on REST architecture**.
- **SOAP Web Services**
  - Web services that **use SOAP to exchange XML formatted data.**

"Remaining slides focus on REST and RESTful services only."

REST : Representational State Transfer

- The **term REST originated in a 2000 doctoral dissertation about the web written by Roy Fielding,**
  - **Dr. Roy Fielding is one of the principal authors of the HTTP protocol specification**

- It has quickly passed into widespread use in the networking community

- **Reference:**
  http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_2

REST : Representational State Transfer

- REST is a **software architectural style for distributed hypermedia systems** like the **World Wide Web**

- **Specifies a set of constraints** (e.g., uniform interface, client/server architecture, stateless communication) **for the architecture of a distributed system**
  - **That, if applied, induce desirable properties, such as performance, scalability, and modifiability**

- In REST architecture style**, data and functionality are considered resources** and are **accessed using URIs**

Key Principles of REST architecture

- **Resource Identification through URI**
  - **Every resource** (which represents data, e.g., customer, order) is **identified by an URI**

- **Uniform Interface**
  - **Uses standard set of HTTP methods**: GET, POST, PUT, DELETE as transport mechanism

- **Multiple Representation of Data**
  - **Supports multiple representation of data** for data exchange: XML, JSON, XHTML, plain text, PDF, JPEG, and others

- **Stateless Communications**
  - **Every interaction with a resource is stateless, i.e., request messages are self-contained**

RESTful Web services

- **Web Services that are implemented based on REST architectural style are referred to as RESTful web services**
- **Uses HTTP verbs** (GET, POST, PUT, DELETE)
- **Expose Resources**, which represent Data, **as an URI**
  - **Each method in a RESTful web service is identified by a unique URL**
  - When the server receives a request, it immediately knows what operation to perform

RESTful Web services (Contd)

- Supports **multiple representation of data** for data exchange: **XML, JSON, XHTML, plain text**
- Emphasizes **stateless communication**
- **Can be used in a program or directly from a web browser**

JAX-RS : Java API for RESTful services

- **Java API that facilitate web services implementation using REST**
- **Provides server-side resource API** (JAX-RS 1.1)
  - Client-side API was added in JAX-RS 1.2
- **Resources : can be modeled as a POJO, JPA Entity, or EJB**
  - **Can be given an ID and parameterized URI**
- **HTTP centric** : uses the notion of URI and HTTP verbs (GET, POST, PUT, DELETE)
- **Data formats**: XML, JSON, XHTML, text

JAX-RS : Java API for RESTful services

- **Annotations driven—Uses annotations to configure a Java class into RESTful web service**
- **Defines resources and actions that can be performed on those resources**
- **Annotations simplify the development of RESTful web services**

JAX-RS : Java API for RESTful services

- **Examples of Annotations specified in JAX-RS**
  - **@Path** : to **specify URI of the resource**
  - **@Produces, @Consumes** : to specify **data format to be sent out** or **received**
  - **@GET, @POST, @PUT, @DELETE** : **to specify which operations to invoke** for these requests
- **Part of Java EE 6 platform and future versions**

Resources

- A simple RESTful web service can be defined as a resource using @Path annotation

```
@Path("orders")
public class OrderResource {
  @GET
  public List<Order> getAll() {
    //. . .
  }

  @GET
  @Path("{oid}")
  public Order getOrder(@PathParam("oid")int id) {
    //. . .
  }
}

@XmlRootElement
public class Order {
  int id;
  //. . .
}
```

Resources: In the previous example

- OrderResource **is a POJO class and is published as a RESTful resource at the** orders **path when we add the class-level** @Path **annotation.**
  - **The** Order **class is marked with the** @XmlRootElement **annotation, allowing a conversion between Java and XML.**
- **The** getAll **resource method, which provides a list of all orders, is invoked when we access this resource using the HTTP GET method;**
  - **we identify it by specifying the @GET annotation on the method.**
- **The** @Path **annotation on the** getOrder **resource method marks it as a subresource that is accessible at** orders/{oid}.
  - **The curly braces around oid identify it as a template parameter and bind its value at runtime to the id parameter of the getOrder resource method.**
  - **The** @PathParam **can also be used to bind template parameters to a resource class field.**

Jersey

- A **reference implementation of JAX-RS**, provided by Sun/Oracle
  - **Implements supports for the annotations defined in JAX-RS spec**

- **Other reference implementations** of JAX-RS specifications:
  - CXF – An **Apache** hosted project is a merger between XFire(a SOAP Framework) and Celtix (an Open Source ESB, sponsored by IONA)
  - RESTEasy– JBoss's JAX-RS project
  - Restlet – This project has been around for a long time and implemented REST before REST was popular. JAX-RS was a natural extension

Creating RESTful Services using Jersey

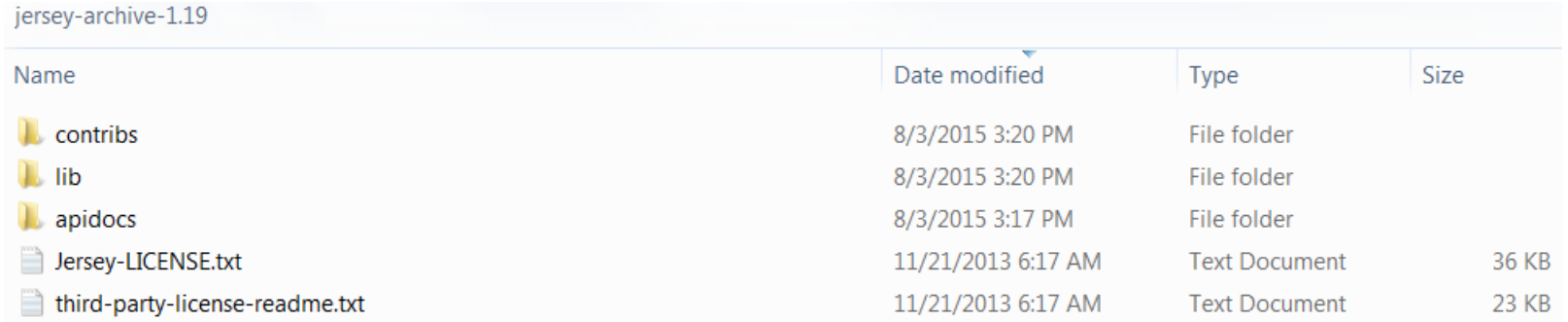
- **Jersey based REST API application is a standard Servlet application**
  - **It has jersey jars in the class path**
  - **Web.xml contains a jersey servlet class ServletContainer, which should be mapped to a URL pattern to handle all REST API requests**

- **Jersey servlet looks at your code (i.e., java class) to determine the method to delegate the HTTP method request to be handled**
  - **This means that http methods need to be mapped to java methods**

Creating RESTful Services using Jersey

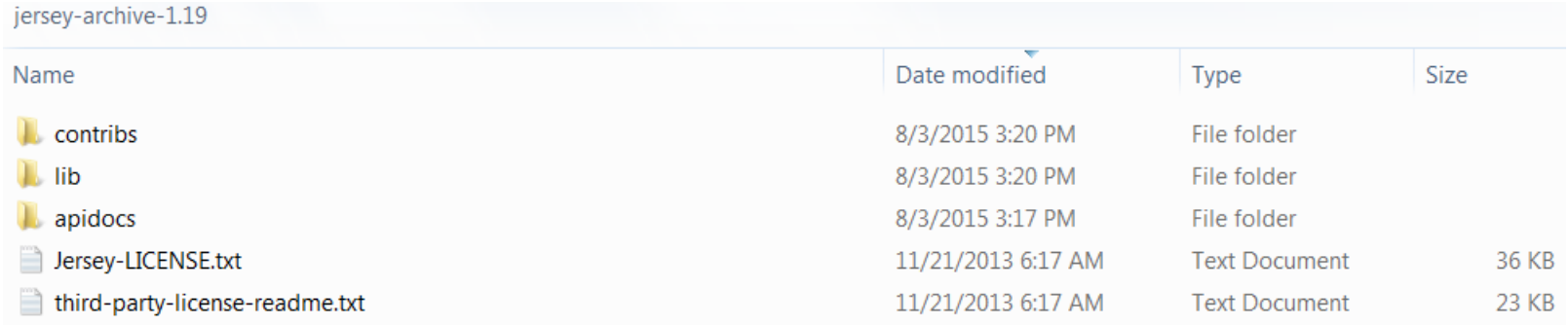
- A **reference implementation of JAX-RS**, provided by Sun/Oracle
  - **Implements supports for the annotations defined in JAX-RS spec**

- **Download Jersey jar files**
  - **Download Jersey zip file from https://jersey.java.net/download.html**
    - **Extracted version of the downloaded zip will look something like this:**

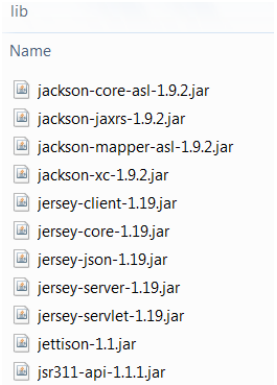


Creating RESTful Services using Jersey

- **Download Jersey zip files – the extracted version will look something like this:**



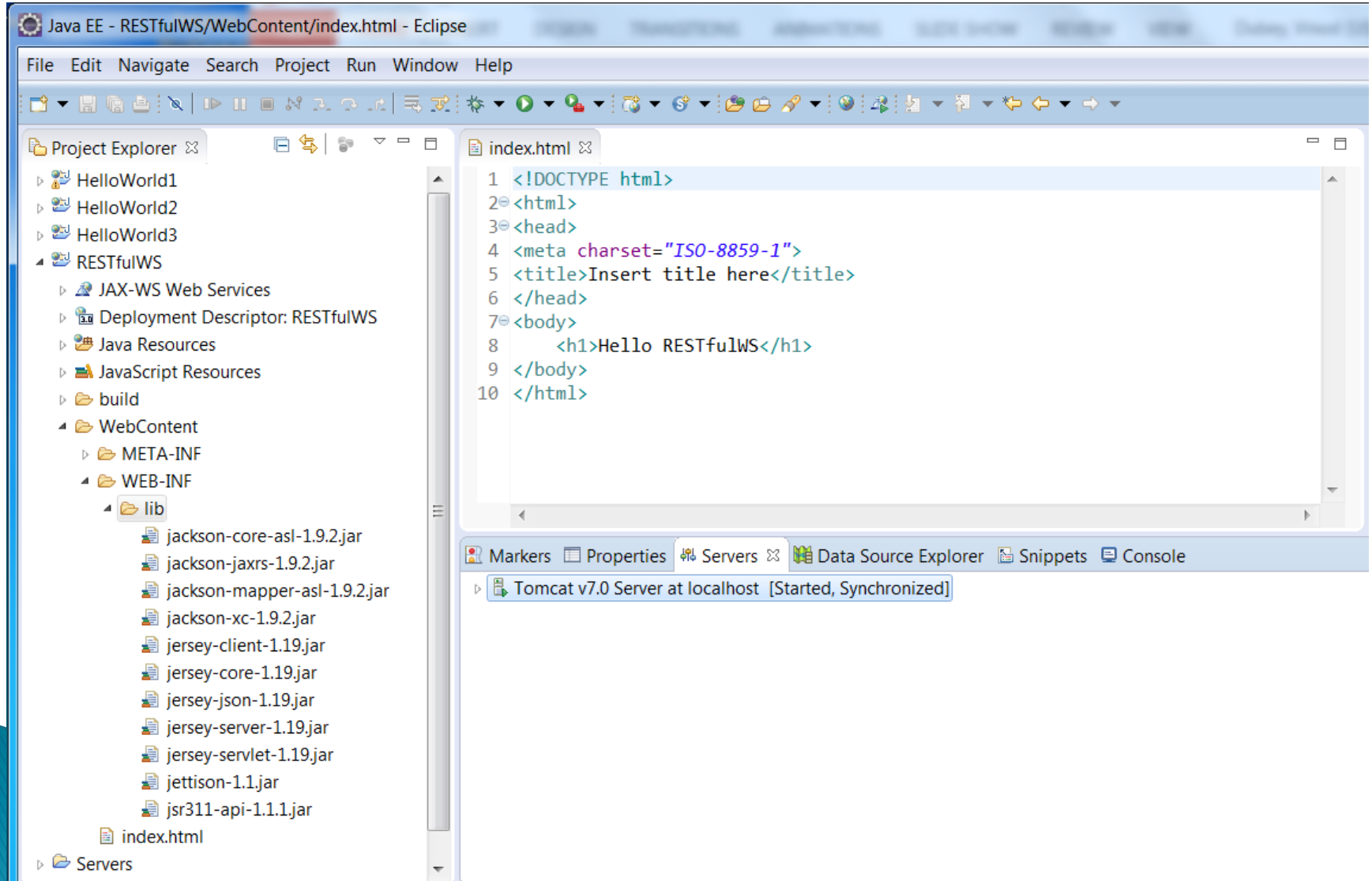
  - **The lib folder (above) contains Jersey jar files and all its dependencies.**



  - **Copy all jars from lib folder (above) to the Web-INF/lib folder of your dynamic web project in Eclipse (Please use Java EE version of Eclipse)**

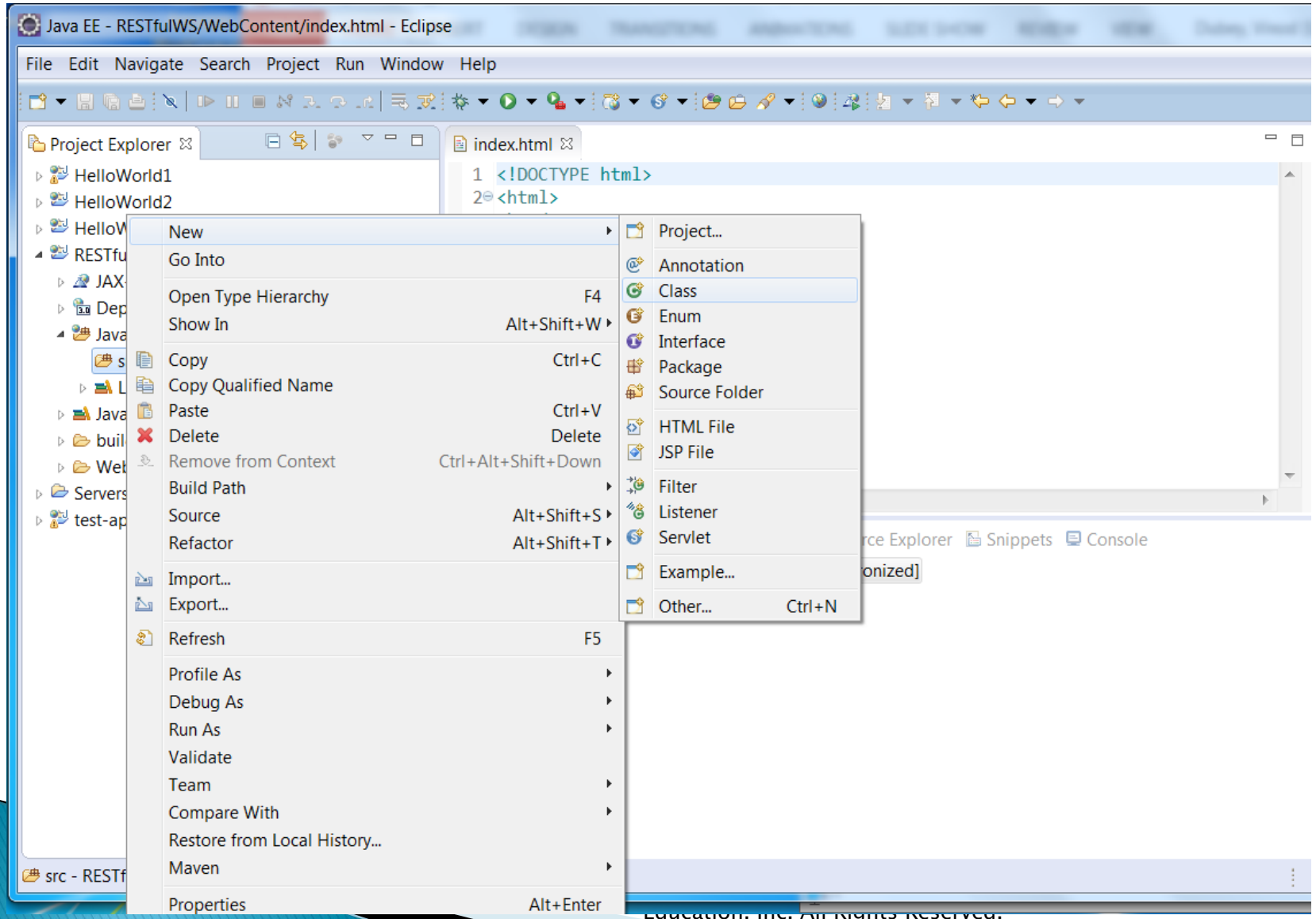
Creating RESTful Services using Jersey

- **Copy jars from lib folder (above) to the WEB-INF/lib folder of your dynamic web project in Eclipse (Java EE version) and put them on Java build path**



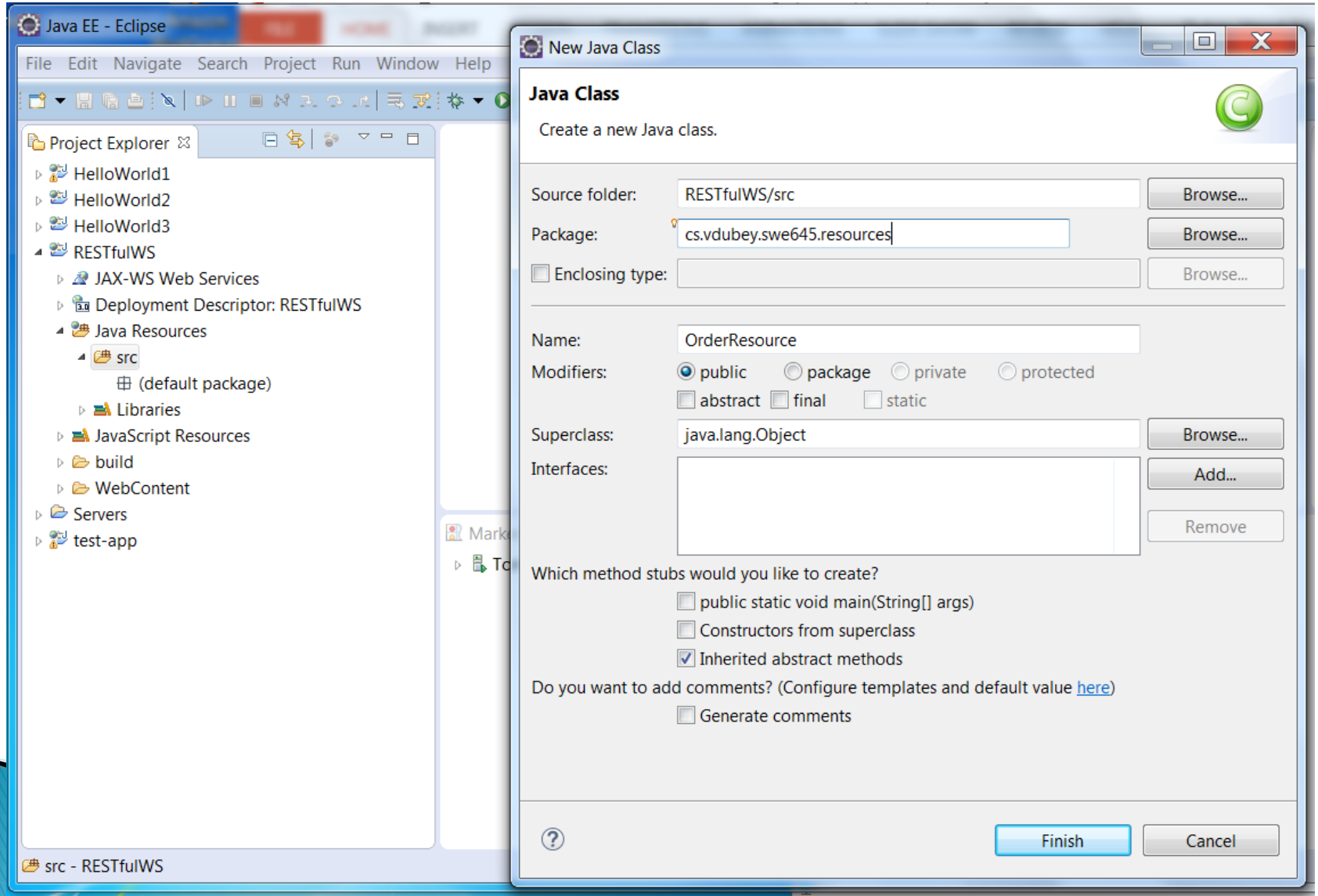
Creating RESTful Services using Jersey

- **R-click on Java Resources/src folder and select New->Class**



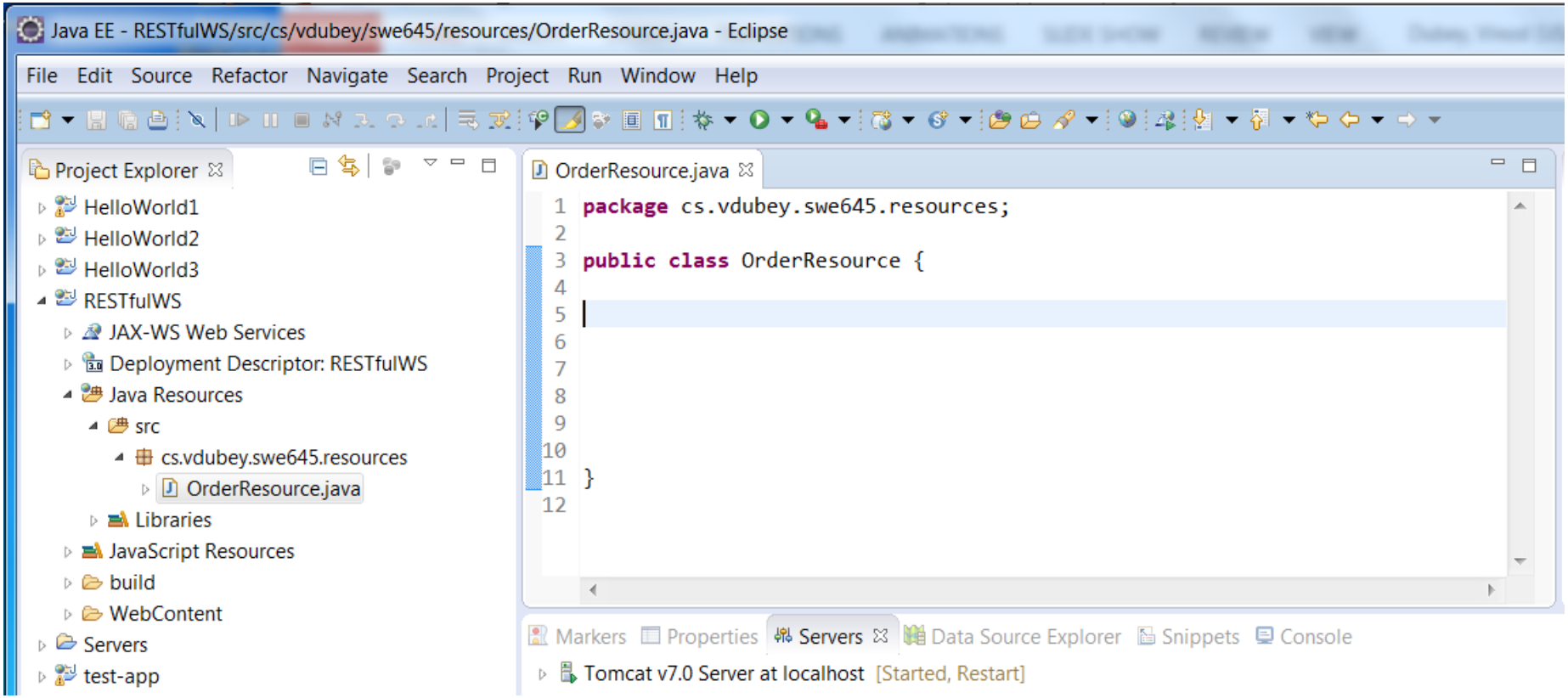
Creating RESTful Services using Jersey

- **Give a package name (e.g., cs.vdubey.swe645.resources) and a name of class as OrderResource and then click Finish**



Creating RESTful Services using Jersey

- **The template of the class OrderResource will be used to add functionality**



Creating RESTful Services using Jersey

- **A simple RESTful web service can be defined as a resource using @Path annotation**

```
@Path("orders")
public class OrderResource {
  @GET
  public List<Order> getAll() {
    //. . .
  }

  @GET
  @Path("{oid}")
  public Order getOrder(@PathParam("oid")int id) {
    //. . .
  }
}

@XmlRootElement
public class Order {
  int id;
  //. . .
}
```

Creating RESTful Services using Jersey

- **Typically, a RESTful resource is bundled in a .war file along with other classes and resources.**

- The JAX-RS provides a deployment agnostic abstract class Application for declaring root resource
  - A Web service may extend this class to declare root resource and provider classes.
  - The Application class also provides additional metadata about the application.

- **The Application class and @ApplicationPath annotation are used to specify the base path for all the RESTful resources in the packaged archive**
  - **Alternatively, use web.xml to specify base path**

Creating RESTful Services using Jersey

- **URL pattern for RESTful requests can be specified in web.xml** (see below) **to indicate that all RESTful requests using /webresource url pattern will be handled by jersey ServletContainer**

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
id="WebApp_ID" version="3.0">


 <servlet>
    <servlet-name>Jersey REST Service</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>Jersey REST Service</servlet-name>
    <url-pattern>/webresources/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Creating RESTful Services using Jersey

- Let's say OrderResource POJO is packaged in the store.war file, deployed at localhost:8080,
  - The Application class is defined as follows:

```
@ApplicationPath("webresources")
public class ApplicationConfig extends Application { }
```

  - OR,
    - /webresources url pattern specified in web.xml, as shown on previous page

- You can access a list of all the orders by issuing a GET request to:

```
@Path("orders")
public class OrderResource {
  @GET
  public List<Order> getAll() {
    //. . .
  }
```

```
http://localhost:8080/store/webresources/orders
```

Creating RESTful Services using Jersey

- **Let's take a look at the following URL used to access the RESTful recourse:**

```
http://localhost:8080/store/webresources/orders
```

- **Please note, when you deploy your web application (e.g., a dynamic web project in eclipse) called store on tomcat, it will be accessible at URL http://localhost:8080/store**
- **The additional url component /webresources/orders is URL to access your RESTful resource, where**
  - **The /webresources appended to http://localhost:8080/store/ indicates that the store web application is configured to accept RESTful requests at /webresources url (it could be any name you configure)**
    - **That is, any RESTful request will go through /webresources uri, which is mapped, in web.xml, to** `org.glassfish.jersey.servlet.ServletContainer`
  - **Anything after http://localhost:8080/webresources/ is the code you write. For example, /orders is the url to your RESTful resource, which is accessible at GET request**

Creating RESTful Services using Jersey

- **Here is the simple RESTful resource defined using @Path annotation that you saw before**

```
@Path("orders")
public class OrderResource {
  @GET
  public List<Order> getAll() {
    //. . .
  }

  @GET
  @Path("{oid}")
  public Order getOrder(@PathParam("oid")int id) {
    //. . .
  }
}

@XmlRootElement
public class Order {
  int id;
  //. . .
}
```

Creating RESTful Services using Jersey

- **The Order class is marked with the @XmlRootElement annotation, allowing a conversion between Java and XML.**

- **The @XmlRootElement annotation enables an automatic mapping from Java to XML following JAXB mapping**
  - **an XML representation of the resource (e.g. a class object) is returned.**

```
@XmlRootElement
public class Order {
   int id;
   //. . .
}
```

Creating RESTful Services using Jersey

- You can obtain a specific order by issuing a GET request to:

```
@Path("orders")
public class OrderResource {

  @GET
  @Path("{oid}")
  public Order getOrder(@PathParam("oid")int id) {
    //. . .
  }
```

```
http://localhost:8080/store/webresources/orders/1
```

- Here, the value 1 will be passed to getOrder's method parameter id.
  - Notice the no use of traditional query string in above URL
- The resource method will locate the order with the correct order number and return back the Order class.

Creating RESTful Services using Jersey

- **A URI may pass HTTP query parameters using name/ value pairs.**
- **You can map these to resource method parameters or fields using the @QueryParam annotation.**
  - If the resource method getAll is updated such that the returned results start from a specific order number, the number of orders returned can also be specified:

```
public List<Order> getAll(@QueryParam("start")int from,
                          @QueryParam("page")int page) {
   //. . .
}
```

- **And the resource is accessed as:**

```
http://localhost:8080/store/webresources/orders?start=10&page=20
```

  - **Then 10 is mapped to the from parameter, and 20 is mapped to the page parameter.**

Binding HTTP Methods

- **JAX-RS provides support for binding standard HTTP GET, POST, PUT, DELETE, HEAD, and OPTIONS methods using the corresponding annotations described below**
  - **You already saw a few examples before**

HTTP methods supported by JAX-RS

HTTP method	JAX-RS annotation
GET	@GET
POST	@POST
PUT	@PUT
DELETE	@DELETE
HEAD	@HEAD
OPTIONS	@OPTIONS

Binding HTTP Methods: @Post example

- **Consider the following HTML form, which takes the order id and customer name and creates an order by posting the form to** *webresources/orders/create*

```
<form method="post" action="webresources/orders/create">
  Order Number: <input type="text" name="id"/><br/>
  Customer Name: <input type="text" name="name"/><br/>
  <input type="submit" value="Create Order"/>
</form>
```

- **The updated resource definition uses the following annotations:**

```
@POST
@Path("create")
@Consumes("application/x-www-form-urlencoded")
public Order createOrder(@FormParam("id")int id,
                         @FormParam("name")String name) {
    Order order = new Order();
    order.setId(id);
    order.setName(name);
    return order;
}
```

Binding HTTP Methods: @POST example

- The @FormParam annotation binds the value of an HTML form parameter to a resource method parameter or a field.
- The value of name attribute in the HTML form and the value of the @FormParam annotation are exactly the same to ensure the binding.
- Clicking the submit button in the form will return the XML representation of the created Order.
  - A Response object may be used to create a custom response

```
@POST
@Path("create")
@Consumes("application/x-www-form-urlencoded")
public Order createOrder(@FormParam("id")int id,
                         @FormParam("name")String name) {
   Order order = new Order();
   order.setId(id);
   order.setName(name);
   return order;
 }
```

```
<form method="post" action="webresources/orders/create">
  Order Number: <input type="text" name="id"/><br/>
  Customer Name: <input type="text" name="name"/><br/>
  <input type="submit" value="Create Order"/>
</form>
```

Binding HTTP Methods: @PUT example

- The resource method is marked as a subresource, and {id} is bound to the resource method parameter id.
- The contents of the body can be any XML media type as defined by @Consumes and are bound to the content method parameter.

```
@PUT
@Path("{id}")
@Consumes("*/xml")
public Order putXml(@PathParam("id")int id,
                    String content) {
  Order order = findOrder(id);
  // update order from "content"
  . . .
  return order;
}
```

- A PUT request to this resource may be issued as

```
curl -i -X PUT -d "New Order"
    http://localhost:8080/store/webresources/orders/1
```

- The content method parameter will have the value New Order.

Binding HTTP Methods: @DELETE example

- An @DELETE resource method can be defined as follows where resource method is marked as a sub-resource, and {id} is bound to the resource method parameter id.

```
@DELETE
@Path("{id}")
public void putXml(@PathParam("id")int id) {
  Order order = findOrder(id);
  // delete order
}
```

- A DELETE request to this resource may be issued as:

```
curl -i -X DELETE
    http://localhost:8080/store/webresources/orders/1
```

Multiple Resource Representation

- By default, a RESTful resource is published or consumed with the */* MIME type.
- A RESTful resource can restrict the media types supported by request and response using the @Consumes and @Produces annotations, respectively.
- These annotations may be specified on the resource class or a resource method.
  - The annotation specified on the method overrides any on the resource class.

Multiple Resource Representation

- **Here is an example showing how Order can be published using multiple MIME types:**

```
@GET
@Path("{oid}")
@Produces({"application/xml", "application/json"})
public Order getOrder(@PathParam("oid")int id) { . . . }
```

- **This resource method can generate an XML or JSON representation of Order.**

- **The exact return type of the response is determined by the HTTP Accept header in the request.**

Multiple Resource Representation

- **Wildcard pattern matching is supported as well.**

- **The following resource method will be dispatched if the HTTP Accept header specifies any application MIME type such as application/xml, application/json, or any other media type:**

```
@GET
@Path("{oid}")
@Produces("application/*")
public Order getOrder(@PathParam("oid")int id) { . . . }
```

Multiple Resource Representation

- An example of how multiple MIME types may be consumed by a resource method.
- The resource method invoked is determined by the HTTP Content-Type header of the request.

```
@POST
@Path("{oid}")
@Consumes({"application/xml", "application/json"})
public Order getOrder(@PathParam("oid")int id) { . . . }
```

Client API

- **JAX-RS 2 adds a new Client API that can be used to access web resources and provides integration with JAX-RS providers.**
- **Uses the fluent builder pattern and creates ClientBuilder object that uses method chaining to build and execute client requests in order to consume the responses returned.**
- **Without this API, users must use a low-level HttpUrlConnection to access the REST endpoint.**

Client API

- To invoke the HTTP GET method by calling the get method.
- The Java type of the response entity is specified as the parameter to the invoked method.
- This code uses the fluent builder pattern and creates ClientBuilder object that uses method chaining to build and execute client requests in order to consume the responses returned.

```
Client client = ClientBuilder.newClient();
Order order = client
   .target("http://localhost:8080/store/webresources/orders")
   .path("{oid}")
   .resolveTemplate("oid", 1)
   .request()
   .get(Order.class);
```

Client API: Steps in the previous example

- **The code uses the fluent builder pattern and works as follows:**
  - **ClientBuilder is the entry point to the client API. It is used to obtain an instance of Client that uses method chaining to build and execute client requests in order to consume the responses returned.**
    - Clients are heavyweight objects that manage the client-side communication infrastructure.
    - Initialization as well as disposal of a Client instance may be a rather expensive operation and thus number of Client instances in the application should be minimized.
  - **We create WebTarget by specifying the URI of the web resource.**
  - We then use these targets to prepare client request invocation by resolving the URI template, using the resolveTemplate method for different names.
  - **We build the client request by invoking the request method.**
  - **We invoke the HTTP GET method by calling the get method.**
  - **The Java type of the response entity is specified as the parameter to the invoked method.**

```
Client client = ClientBuilder.newClient();
Order order = client
   .target("http://localhost:8080/store/webresources/orders")
   .path("{oid}")
   .resolveTemplate("oid", 1)
   .request()
   .get(Order.class);
```

Client API

- We can make HTTP POST or PUT requests by using the post or put methods, respectively

- In this code below, a new message entity is created with the specified media type, a POST request is created, and a response of type Order is expected.

```
Order order =
  client
    .target(...)
    .request()
    .post(Entity.entity(new Order(1), "application/json"),
      Order.class);
```

Client API

- We can make an HTTP DELETE request by identifying the resource with the URI and using the delete method:

```
client
   .target("...")
   .target("{oid}")
   .resolveTemplate("oid", 1)
   .request()
   .delete();
```

JPA Entity exposed as a RESTful Resource

- Next example:
  - A JPA Entity class can be configured and used as a RESTful Web service

A JPA Entity class: Book

```
package entities;

import java.io.Serializable;
import javax.persistence.*;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import javax.xml.bind.annotation.XmlRootElement;

/**
 *
 * @author vinod
 */
@Entity
@Table(name = "BOOK")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Book.findAll", query = "SELECT b FROM Book b"),
    @NamedQuery(name = "Book.findByBookId", query = "SELECT b FROM Book b WHERE b.bookId = :bookId"),
    @NamedQuery(name = "Book.findByTitle", query = "SELECT b FROM Book b WHERE b.title = :title"),
    @NamedQuery(name = "Book.findByAuthor", query = "SELECT b FROM Book b WHERE b.author = :author"),
    @NamedQuery(name = "Book.findByPrice", query = "SELECT b FROM Book b WHERE b.price = :price")})
public class Book implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @NotNull
    @Size(min = 1, max = 300)
    @Column(name = "BOOK_ID")
    private String bookId;
    @Size(max = 300)
    @Column(name = "TITLE")
    private String title;
    @Size(max = 300)
    @Column(name = "AUTHOR")
    private String author;
```

```
@Column(name = "PRICE")
private Float price;

public Book() {
}

public Book(String bookId) {
    this.bookId = bookId;
}

public String getBookId() {
    return bookId;
}

public void setBookId(String bookId) {
    this.bookId = bookId;
}

public String getTitle() {
    return title;
}

public void setTitle(String title) {
    this.title = title;
}

public String getAuthor() {
    return author;
}

public void setAuthor(String author) {
    this.author = author;
}
```

```java
    public Float getPrice() {
        return price;
    }

    public void setPrice(Float price) {
        this.price = price;
    }

    @Override
    public int hashCode() {
        int hash = 0;
        hash += (bookId != null ? bookId.hashCode() : 0);
        return hash;
    }

    @Override
    public boolean equals(Object object) {
        // TODO: Warning - this method won't work in the case the id fields are not set
        if (!(object instanceof Book)) {
            return false;
        }
        Book other = (Book) object;
        if ((this.bookId == null && other.bookId != null) || (this.bookId != null && !this.bookId.equals(other.bookId))) {
            return false;
        }
        return true;
    }

    @Override
    public String toString() {
        return "entities.Book[ bookId=" + bookId + " ]";
    }

}
```

BookFacadeREST: a Façade class as a wrapper to JPA Entity Book deployed as a RESTful Resource

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package service;

import entities.Book;
import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.ws.rs.*;

/**
 *
 * @author vinod
 */
@Stateless
@Path("book")
public class BookFacadeREST extends AbstractFacade<Book> {
    @PersistenceContext(unitName = "BookStorePU")
    private EntityManager em;

    public BookFacadeREST() {
        super(Book.class);
    }

    @POST
    @Override
    @Consumes({"application/xml", "application/json"})
    public void create(Book entity) {
        super.create(entity);
    }
```

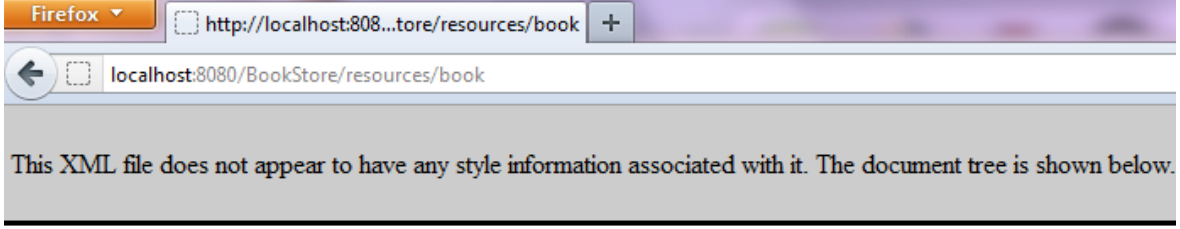
```java
@PUT
@Override
@Consumes({"application/xml", "application/json"})
public void edit(Book entity) {
    super.edit(entity);
}

@DELETE
@Path("{id}")
public void remove(@PathParam("id") String id) {
    super.remove(super.find(id));
}

@GET
@Path("{id}")
@Produces({"application/xml", "application/json"})
public Book find(@PathParam("id") String id) {
    return super.find(id);
}

@GET
@Override
@Produces({"application/xml", "application/json"})
public List<Book> findAll() {
    return super.findAll();
}

@GET
@Path("{from}/{to}")
@Produces({"application/xml", "application/json"})
public List<Book> findRange(@PathParam("from") Integer from, @PathParam("to") Integer to) {
    return super.findRange(new int[]{from, to});
}
```

```
    @GET
    @Path("count")
    @Produces("text/plain")
    public String countREST() {
        return String.valueOf(super.count());
    }

    @java.lang.Override
    protected EntityManager getEntityManager() {
        return em;
    }

}
```

AbstractFaçade class as a wrapper to a JPA Entity

```
package service;

import java.util.List;
import javax.persistence.EntityManager;

/**
 *
 * @author vinod
 */
public abstract class AbstractFacade<T> {
    private Class<T> entityClass;

    public AbstractFacade(Class<T> entityClass) {
        this.entityClass = entityClass;
    }

    protected abstract EntityManager getEntityManager();

    public void create(T entity) {
        getEntityManager().persist(entity);
    }

    public void edit(T entity) {
        getEntityManager().merge(entity);
    }

    public void remove(T entity) {
        getEntityManager().remove(getEntityManager().merge(entity));
    }

    public T find(Object id) {
        return getEntityManager().find(entityClass, id);
    }
```

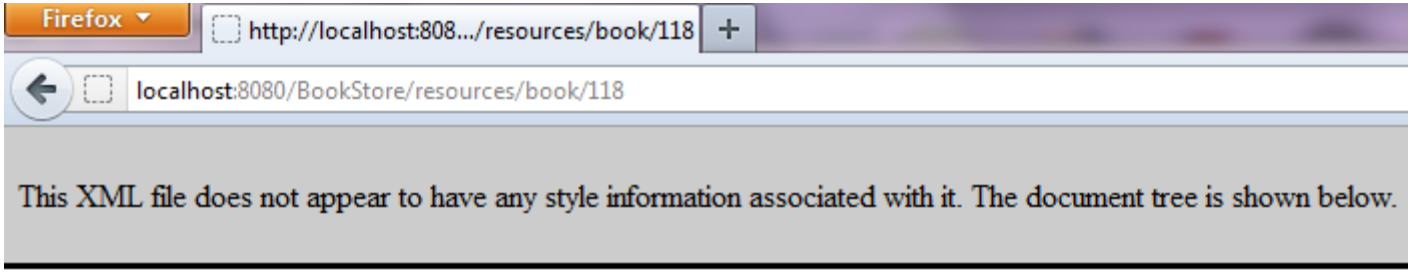
```
    public T find(Object id) {
        return getEntityManager().find(entityClass, id);
    }

    public List<T> findAll() {
        javax.persistence.criteria.CriteriaQuery cq = getEntityManager().getCriteriaBuilder().createQuery();
        cq.select(cq.from(entityClass));
        return getEntityManager().createQuery(cq).getResultList();
    }

    public List<T> findRange(int[] range) {
        javax.persistence.criteria.CriteriaQuery cq = getEntityManager().getCriteriaBuilder().createQuery();
        cq.select(cq.from(entityClass));
        javax.persistence.Query q = getEntityManager().createQuery(cq);
        q.setMaxResults(range[1] - range[0]);
        q.setFirstResult(range[0]);
        return q.getResultList();
    }

    public int count() {
        javax.persistence.criteria.CriteriaQuery cq = getEntityManager().getCriteriaBuilder().createQuery();
        javax.persistence.criteria.Root<T> rt = cq.from(entityClass);
        cq.select(getEntityManager().getCriteriaBuilder().count(rt));
        javax.persistence.Query q = getEntityManager().createQuery(cq);
        return ((Long) q.getSingleResult()).intValue();
    }

}
```




```
− <books>
  − <book>
      <author>John Doe</author>
      <bookId>100</bookId>
      <price>19.99</price>
      <title>Java EE for Dummies</title>
    </book>
  − <book>
      <author>Jane Doe</author>
      <bookId>101</bookId>
      <price>29.99</price>
      <title>Java EE in a nutshell</title>
    </book>
  − <book>
      <author>Jane Doe</author>
      <bookId>102</bookId>
      <price>39.99</price>
      <title>EJB 3.0 in a nutshell</title>
    </book>
  − <book>
      <author>Larry Stark</author>
      <bookId>103</bookId>
      <price>24.99</price>
      <title>All you need know about Java</title>
    </book>
  − <book>
      <author>Mark Tripputti</author>
      <bookId>104</bookId>
      <price>24.99</price>
      <title>SOAP Web Services</title>
    </book>
  − <book>
      <author>Mark Tripputti</author>
      <bookId>105</bookId>
      <price>24.99</price>
```


Accessing RESTful service directly from the browser

```
- <book>
    <author>Mark Tripputti</author>
    <bookId>105</bookId>
    <price>24.99</price>
    <title>RESTful Web Services</title>
  </book>
- <book>
    <author>Kendra Singer</author>
    <bookId>106</bookId>
    <price>34.99</price>
    <title>JPA 2.0</title>
  </book>
- <book>
    <author>David Hafflefinger</author>
    <bookId>107</bookId>
    <price>34.99</price>
    <title>JSF 2.0</title>
  </book>
- <book>
    <author>Michael Brundage</author>
    <bookId>108</bookId>
    <price>54.99</price>
    <title>XQuery The XML Query Language</title>
  </book>
- <book>
    <author>Ramesh Loganathan</author>
    <bookId>109</bookId>
    <price>54.99</price>
    <title>SOA Approach to Integration</title>
  </book>
- <book>
    <author>Michael Brundage</author>
    <bookId>110</bookId>
    <price>54.99</price>
    <title>XQuery The XML Query Language</title>
  </book>
- <book>
    <author>Kai Hwang and Jack Dongarra</author>
    <bookId>111</bookId>
    <price>54.99</price>
```




```
−<book>
    <author>Peter Doe</author>
    <bookId>118</bookId>
    <price>39.99</price>
    <title>HTML 5 : All you need to know</title>
  </book>
```


RESTful service accessed directly from the browser

Uses a parameter!

Useful resources on RESTful Web service with Java (JAX-RS) using Jersey

- http://www.vogella.com/tutorials/REST/article.html
- https://www.tutorialspoint.com/restful/

Backup

Creating a REST-Based XML Web Service: Some more examples

- **Any IDEs such as NetBeans or Eclipse** can be used to create RESTful Web services, **but** you should be able to create them by **creating a dynamic web project** and **using annotations on your class**

- IDEs can provide user interface to generate the class and set up the proper annotations.

Creating a REST-Based XML Web Service (cont.)

- The `@Path` annotation on the `WelcomeRESTXMLResource` class indicates the URI for accessing the web service.
  - This is **appended** to the **web application project's URL** to invoke the service.

```
1  // Fig. 31.12: WelcomeRESTXMLResource.java
2  // REST web service that returns a welcome message as XML.
3  package com.deitel.welcomerestxml;
4
5  import java.io.StringWriter;
6  import javax.ws.rs.GET; // annotation to indicate method uses HTTP GET
7  import javax.ws.rs.Path; // annotation to specify path of resource
8  import javax.ws.rs.PathParam; // annotation to get parameters from URI
9  import javax.ws.rs.Produces; // annotation to specify type of data
10 import javax.xml.bind.JAXB; // utility class for common JAXB operations
11
12 @Path( "welcome" ) // URI used to access the resource
```

Fig. 31.12 | REST web service that returns a welcome message as XML. (Part 1 of 2.)

```
13  public class WelcomeRESTXMLResource
14  {
15     // retrieve welcome message
16     @GET // handles HTTP GET requests
17     @Path( "{name}" ) // URI component containing parameter
18     @Produces( "application/xml" ) // response formatted as XML
19     public String getXml( @PathParam( "name" ) String name )
20     {
21        String message = "Welcome to JAX-RS web services with REST and " +
22           "XML, " + name + "!"; // our welcome message
23        StringWriter writer = new StringWriter();
24        JAXB.marshal( message, writer ); // marshal String as XML
25        return writer.toString(); // return XML as String
26     } // end method getXml
27  } // end class WelcomeRESTXMLResource
```

Fig. 31.12 | REST web service that returns a welcome message as XML. (Part 2 of 2.)

Creating a REST-Based XML Web Service (cont.)

- **Methods** of the class can **also use the `@Path` annotation**.
  - Parts of the **path** specified in **curly braces** indicate **parameters**—
  - they are placeholders for values that are passed to the web service as part of the path.
- Arguments in a URL can be used as arguments to a web service method.
  - To do so, you bind the parameters specified in the `@Path` specification to parameters of the web service method with the `@PathParam` annotation.
  - When the request is received, the server passes the argument(s) in the URL to the appropriate parameter(s) in the web service method.

Creating a REST-Based XML Web Service (cont.)

- The **`@GET` annotation** denotes that this method is accessed via an HTTP `GET` request.
- The **`@Produces` annotation** denotes the content type returned to the client.
  - It is **possible to have multiple methods with the same HTTP method and path but different `@Produces` annotations**, and JAX-RS will call the method matching the content type requested by the client.
- The **`@Consumes` annotation** restricts the content type that the web service will accept from a client.

Creating a REST-Based XML Web Service (cont.)

- JAXB class from package `javax.xml.bind`.
  - JAXB (Java Architecture for XML Binding) is a set of classes for converting POJOs to and from XML.
  - `JAXB` class contains easy-to-use wrappers for common operations.
- `JAXB` `static` method marshal converts its argument to XML format.

Consuming a REST-Based XML Web Service

- RESTful web services do not require web service references.

Consuming a REST-Based XML Web Service

```
1  // Fig. 31.14: WelcomeRESTXMLClientJFrame.java
2  // Client that consumes the WelcomeRESTXML service.
3  package com.deitel.welcomerestxmlclient;
4
5  import javax.swing.JOptionPane;
6  import javax.xml.bind.JAXB; // utility class for common JAXB operations
7
8  public class WelcomeRESTXMLClientJFrame extends javax.swing.JFrame
9  {
10    // no-argument constructor
11    public WelcomeRESTXMLClientJFrame()
12    {
13       initComponents();
14    } // end constructor
15
16    // The initComponents method is autogenerated by NetBeans and is called
17    // from the constructor to initialize the GUI. This method is not shown
18    // here to save space. Open WelcomeRESTXMLClientJFrame.java in this
19    // example's folder to view the complete generated code.
20
```

Fig. 31.14 | Client that consumes the `WelcomeRESTXML` service. (Part 1 of 4.)

```
21      // call the web service with the supplied name and display the message
22      private void submitJButtonActionPerformed(
23         java.awt.event.ActionEvent evt)
24      {
25         String name = nameJTextField.getText(); // get name from JTextField
26
27         // the URL for the REST service
28         String url =
29            "http://localhost:8080/WelcomeRESTXML/resources/welcome/" + name;
30
31         // read from URL and convert from XML to Java String
32         String message = JAXB.unmarshal( url, String.class );
33
34         // display the message to the user
35         JOptionPane.showMessageDialog( this, message,
36               "Welcome", JOptionPane.INFORMATION_MESSAGE );
37      } // end method submitJButtonActionPerformed
38
```

Fig. 31.14 | Client that consumes the `WelcomeRESTXML` service. (Part 2 of 4.)

```
39      // main method begins execution
40      public static void main( String args[] )
41      {
42         java.awt.EventQueue.invokeLater(
43            new Runnable()
44            {
45               public void run()
46               {
47                  new WelcomeRESTXMLClientJFrame().setVisible( true );
48               } // end method run
49            } // end anonymous inner class
50         ); // end call to java.awt.EventQueue.invokeLater
51      } // end main
52
53      // Variables declaration - do not modify
54      private javax.swing.JLabel nameJLabel;
55      private javax.swing.JTextField nameJTextField;
56      private javax.swing.JButton submitJButton;
57      // End of variables declaration
58   } // end class WelcomeRESTXMLClientJFrame
```

Fig. 31.14 | Client that consumes the `WelcomeRESTXML` service. (Part 3 of 4.)

Fig. 31.14 | Client that consumes the `WelcomeRESTXML` service. (Part 4 of 4.)

Publishing and Consuming REST-Based JSON Web Services

- **JSON is designed as a *data exchange format*.**
  - An **alternative way (to XML)** to pass data between the client and the server
  - Represented as a list of **property names** and **values** (**both in double quotation marks) contained in curly braces**
  - **Array** represented in JSON **with square brackets** containing a **comma-separated list of values**
  - Each value in a JSON array can be a string, a number, a JSON representation of an object, `true`, `false` or `null`

```
1 [ { "first": "Cheryl", "last": "Black" },
2   { "first": "James", "last": "Blue" },
3   { "first": "Mike", "last": "Brown" },
4   { "first": "Meg", "last": "Gold" } ]
```

Publishing and Consuming REST-Based JSON Web Services

- There are many **open-source JSON libraries for Java** and other languages;
  - you can find a list of them at **`json.org`**.
- The following slides use the Gson library from `code.google.com/p/google-gson/`.

```
1  // Fig. 31.15: WelcomeRESTJSONResource.java
2  // REST web service that returns a welcome message as JSON.
3  package com.deitel.welcomerestjson;
4
5  import com.google.gson.Gson; // converts POJO to JSON and back again
6  import javax.ws.rs.GET; // annotation to indicate method uses HTTP GET
7  import javax.ws.rs.Path; // annotation to specify path of resource
8  import javax.ws.rs.PathParam; // annotation to get parameters from URI
9  import javax.ws.rs.Produces; // annotation to specify type of data
10
11 @Path( "welcome" ) // path used to access the resource
12 public class WelcomeRESTJSONResource
13 {
14    // retrieve welcome message
15    @GET // handles HTTP GET requests
16    @Path( "{name}" ) // takes name as a path parameter
17    @Produces( "application/json" ) // response formatted as JSON
18    public String getJson( @PathParam( "name" ) String name )
19    {
20       // add welcome message to field of TextMessage object
21       TextMessage message = new TextMessage(); // create wrapper object
22       message.setMessage( String.format( "%s, %s!",
23          "Welcome to JAX-RS web services with REST and JSON", name ) );
```

Fig. 31.15 | REST web service that returns a welcome message as JSON. (Part 1 of 2.)

```
24
25          return new Gson().toJson( message ); // return JSON-wrapped message
26       } // end method getJson
27    } // end class WelcomeRESTJSONResource
28
29    // private class that contains the message we wish to send
30    class TextMessage
31    {
32       private String message; // message we're sending
33
34       // returns the message
35       public String getMessage()
36       {
37          return message;
38       } // end method getMessage
39
40       // sets the message
41       public void setMessage( String value )
42       {
43          message = value;
44       } // end method setMessage
45    } // end class TextMessage
```

Fig. 31.15 | REST web service that returns a welcome message as JSON. (Part 2 of 2.)

Consuming a REST-Based JSON Web Service

- **URL method openStream can invoke the web service** and obtains an `InputStream` from which the client can read the response.

```
1  // Fig. 31.16: WelcomeRESTJSONClientJFrame.java
2  // Client that consumes the WelcomeRESTJSON service.
3  package com.deitel.welcomerestjsonclient;
4
5  import com.google.gson.Gson; // converts POJO to JSON and back again
6  import java.io.InputStreamReader;
7  import java.net.URL;
8  import javax.swing.JOptionPane;
9
10 public class WelcomeRESTJSONClientJFrame extends javax.swing.JFrame
11 {
12    // no-argument constructor
13    public WelcomeRESTJSONClientJFrame()
14    {
15       initComponents();
16    } // end constructor
17
```

Fig. 31.16 | Client that consumes the `WelcomeRESTJSON` service. (Part 1 of 5.)

```
18      // The initComponents method is autogenerated by NetBeans and is called
19      // from the constructor to initialize the GUI. This method is not shown
20      // here to save space. Open WelcomeRESTJSONClientJFrame.java in this
21      // example's folder to view the complete generated code.
22
23      // call the web service with the supplied name and display the message
24      private void submitJButtonActionPerformed(
25         java.awt.event.ActionEvent evt )
26      {
27         String name = nameJTextField.getText(); // get name from JTextField
28
```

Fig. 31.16 | Client that consumes the `WelcomeRESTJSON` service. (Part 2 of 5.)

```
29          // retrieve the welcome string from the web service
30          try
31          {
32             // the URL of the web service
33             String url = "http://localhost:8080/WelcomeRESTJSON/" +
34                "resources/welcome/" + name;
35
36             // open URL, using a Reader to convert bytes to chars
37             InputStreamReader reader =
38                new InputStreamReader( new URL( url ).openStream() );
39
40             // parse the JSON back into a TextMessage
41             TextMessage message =
42                new Gson().fromJson( reader, TextMessage.class );
43
44             // display message to the user
45             JOptionPane.showMessageDialog( this, message.getMessage(),
46                "Welcome", JOptionPane.INFORMATION_MESSAGE );
47          } // end try
48          catch ( Exception exception )
49          {
50             exception.printStackTrace(); // show exception details
51          } // end catch
52       } // end method submitJButtonActionPerformed
```

Fig. 31.16 | Client that consumes the `WelcomeRESTJSON` service. (Part 3 of 5.)

```
53
54      // main method begin execution
55      public static void main( String args[] )
56      {
57         java.awt.EventQueue.invokeLater(
58            new Runnable()
59            {
60               public void run()
61               {
62                  new WelcomeRESTJSONClientJFrame().setVisible( true );
63               } // end method run
64            } // end anonymous inner class
65         ); // end call to java.awt.EventQueue.invokeLater
66      } // end main
67
68      // Variables declaration - do not modify
69      private javax.swing.JLabel nameJLabel;
70      private javax.swing.JTextField nameJTextField;
71      private javax.swing.JButton submitJButton;
72      // End of variables declaration
73   } // end class WelcomeRESTJSONClientJFrame
74
```

Fig. 31.16 | Client that consumes the `WelcomeRESTJSON` service. (Part 4 of 5.)

```
75  // private class that contains the message we are receiving
76  class TextMessage
77  {
78     private String message; // message we're receiving
79
80     // returns the message
81     public String getMessage()
82     {
83        return message;
84     } // end method getMessage
85
86     // sets the message
87     public void setMessage( String value )
88     {
89        message = value;
90     } // end method setMessage
91  } // end class TextMessage
```

Fig. 31.16 | Client that consumes the `WelcomeRESTJSON` service. (Part 5 of 5.)

When to use RESTful Web services

- REST is ideally suited for
  - Integration over the Web
  - Exposing data over the Internet as well as
  - Combining data from different resources in a Web browser

- A common use
  - Act as a frontend to a database
  - RESTful client can use RESTful Web services to perform CRUD operations in a database

When to use RESTful Web services

- Use RESTful when the web service is stateless
- The service producer and service consumer have a mutual understanding of the context and content being passed along
- Limited bandwidth

When to use SOAP-based Web services

- A **formal contract** must be established to **describe the interface** that the web service offers
- Suited for **enterprise application integration** that have **advanced QoS requirements**
- The **architecture must address nonfunctional requirements** (Supported by WS*….)
- More on SOAP web services later……