Contents

[Introduction 2](#_Toc135418392)

[Developing a JAX-RS RESTful Service 2](#_Toc135418393)

[Customer: The Data Class 2](#_Toc135418394)

[CustomerResource: Our JAX-RS Service 3](#_Toc135418395)

[Creating Customers 4](#_Toc135418396)

[Retrieving Customers 4](#_Toc135418397)

[Updating A Customer 5](#_Toc135418398)

[Utility Methods 5](#_Toc135418399)

[JAX-RS And Java Interfaces 7](#_Toc135418400)

[Inheritance 8](#_Toc135418401)

# Introduction

1. 1st two chapters focused on.
   1. REST Theory.
   2. REST Design.
2. It’s time to implement a part of our system in Java Language.
3. RESTful Web Services became possible with Servlets.  
   But in 2008, a new specification called JAX-RS was to simplify RESTful Service Implementation as in Servlet a lot of boilerplate code.
4. **Definition**: JAX-RS is a framework that focus on applying Java Annotations to plain Java Objects.
   1. It has **annotations** to bind a specific URI and HTTP operation to an individual method of your Java Class.
   2. It has **parameter injection annotations** to easily pull in information from request.
   3. It has **message body readers** and **writers** that allow you to decouple data format **marshalling** and **unmarshalling** from your **java data objects**.
   4. It has exception mappers that can map an application-thrown exception to an **HTTP response code and message**.
   5. Finally, it has some nice facilities for **HTTP Content Negotiation**.
5. This chapter gives a brief introduction to writing a JAX-RS service.  
   You will find that getting it up and running it is fairly simple.

# Developing a JAX-RS RESTful Service

1. Let’s start by implementing one of the resources of the order entry system we define in chapter 2.  
   Specifically, we will define a **JAX-RS Service** that allows us to read, create, update Customers.
2. To do this, we need to implement two java classes.
   1. One class will be used to represent **actual Customers**.
   2. The other will be our **JAX-RS Service**.

## Customer: The Data Class

1. We need java class to represent Customers in our system named **Customer.java**.
2. **The definition**:  
   A white background with red text

   Description automatically generated with low confidence
3. In Enterprise Java App, The Customer class would usually be a **JPA Entity Bean** and would be used to interact with a relational Database.  
   It could also be annotated with **JAXB annotations** that allows you to map a **java class directly to XML**.  
   But to keep our example simple, Customer will be a plain java object and stored in memory.
4. In Chapter 6, we will learn how to make translating b/w your customer’s data format (XML) and your Custom Objects easier.  
   Chapter 14 will show you how JAX-RS works in context of a Java EE Application and things like JPA.

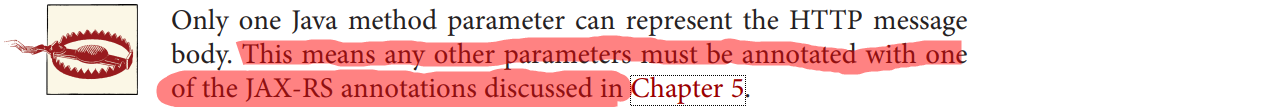
## CustomerResource: Our JAX-RS Service

1. Now we have identified our **Domain Objects** that will represent our customers at runtime, we need to implement our **JAX-RS Service** so that remote clients can interact with our **Customer Database**.
2. **Definition**: JAX-RS Service is a Java Class that uses JAX-RS annotations to bind and map specific incoming HTTP requests to Java Methods that can service these requests.
3. While **JAX-RS** can be integrated with popular component models like **EJB, Web Beans, JBoss Seam, and Spring**, it does define its own **lightweight model**.
4. In vanilla JAX-RS, services can be **singletons** or **per-request** objects.
   1. **A Singleton** means that one and only one Java Object services HTTP requests.
   2. **Per-Request** means that a Java Object is created to process each incoming request and is thrown away at the end of the request.
      1. **Per-Request also** implies **statelessness**, as no service state is held between requests.
5. For our example, we will write a **CustomerResource class** to **implement our JAX-RS service** and assume it will be a **singleton** as it is going to hold **state** 🡺 Map of Customer Objects.
6. Let’s see how to write **JAX-RS Service**.  
   A picture containing text, screenshot, font, line

   Description automatically generated
   1. It is plain Java Class as it is not implementing/extending any JAX-RS interface/class.
   2. **@javax.ws.rs.Path** on class **CustomerResource.java** designates this class as a **JAX-RS service**.
   3. **Note** that the value of **@Path(“/customers”)** which is “/customers” represents the **relative Root URI** of our Customer service.   
      If the absolute base URI of our server is <http://shop.restfully.com>, the methods exposed by the CustomerResource class will be available under <http://shop.restfully.com/customers>.
   4. Declared properties under class CustomerResource.java for storing customer in memory and creating ID for a new customer.
      1. private Map<Integer, Customer> customerDB = **new ConcurrentHashMap**<Integer, Customer>();
         1. As our JAX-RS resource CustomerResource is singleton so multiple concurrent requests will be accessing the Map.
         2. **HashMap** will create **current access exception**.
         3. Hashtable will create **synchronization bottleneck.**
      2. private AtomicInteger idCounter = new AtomicInteger();
         1. To create IDs for newly generated customers.
         2. Due to concurrent access, we used AtomicInteger.

# Creating Customers

1. Let’s see how to create customers in our CustomerResource Class.
2. A screen shot of a computer code

   Description automatically generated with low confidence
   1. We will implement customer creation using the same model as that used in Chapter 2.   
      An **HTTP POST request** sends the **XML document** representing the Customer we want to create.
   2. createCustomer(InputStream is) receives the requests, parse the document to Customer object and stores into customerDB.  
      The method returns the response code 201 “Created”, along with a Location header pointing to the absolute URI of the customer we just created.
   3. How all this is done by createCustomer(InputStream is), let’s discuss.
   4. Binding HTTP POST request with URI relative /customers.
      1. We put **@@javax.ws.rs.Path(“/customers”)** on CustomerResource.java and   
         **@javax.ws.rs.POST** annotation on createCustomer(InputStream is) which together binds **HTTP POST** request with relative URI “/customers” to the method createCustomer(InputStream is).
   5. **@javax.ws.rs.Consumes 🡺 Media Type Expectation**:
      1. This annotation specifies the **media type** in the **message body** (Including headers) of the **HTTP input request**.
      2. If media type other than XML, then an error code is sent back to the client.
   6. The **method parameter** in method **createCustomer(InputStream is)**:
      1. In JAX-RS, any non-JAX-RS-annotated parameter is considered to be a representation of the **HTTP input request’s message body**.
      2. In this case, we want access to the message body in its most basic form, an **InputStream**.  
         
   7. javax.ws.rs.core.Response:
      1. **Response.created()**: Creates a Response object with HTTP Status code 201, “Created”.
      2. **URI.create(“/customers/”, customer.getId())).build()**
         1. Creates an absolute URI of the newly created customer something like  
            <http://shop.restfully.com/customers/333> depending on the base URI of the server and customer id.

## Retrieving Customers

1. d
   1. **@javax.ws.rs.GET** annotation on getCustomer(@PathParam(“id”) int id), binds HTTP GET request to this method with relative URI “/customers” which we defined on CustomerResource like this @Path(“/customers”).
   2. **@javax.ws.rs.Produces(“application/xml”)**:
      1. It tells JAX-RS which HTTP **Content-Type** the GET response will be.  
         In this case 🡺 “**application/xml”.**
   3. If a customer with a given ID is not found in customerDB Map, we throw **javax.ws.rs.WebApplicationException**.
      1. This exception will set HTTP Status Code to **404, “Not Found”**.
      2. We will discuss more about this in **Chapter 7**.
   4. We will write the response back to the client through a **java.io.OutputStream**.
      1. In JAX-RS, when you want to do steaming manually, you must implement and return an instance of the **javax.ws.rs.core.SteamingOutput** interface from your **JAX-RS method (meaning annotated with JAX-RS Methods like @POST)**.
      2. That interface has one callback method **javax.ws.rs.core.SteamingOutput.write(OutputStream).**
      3. In above slide, we implement and return an inner class implementation of **javax.ws.rs.core.SteamingOutput.**
      4. Within the write(OutputStream), we’re calling our utility method called **outputCustomer**() that exits in CustomerResource.java itself.
      5. When the **JAX-RS Provider** is ready to send response back to the client, it will call **javax.ws.rs.core.SteamingOutput.write(OutputStream) method** that we implemented to output XML representation of the requested customer**.**
      6. In general, we will not use **javax.ws.rs.core.SteamingOutput** to output responses.  
         In **Chapter 6**, we will see that JAX-RS has a bunch of nice **Content Handlers** thatcan automatically convert Java Objects straight into the Data Format we’re sending across the wire.

## Updating A Customer

1. This is our last RESTful Operation.
2. In Chapter 2, we used **PUT /customers/{id},** while passing along an updated XML representation of the customer.
3. This is implemented in the **updateCustomer()** method of our **CustomerReresource** class.
4. A screenshot of a computer program

   Description automatically generated with low confidence
   1. **@javax.ws.rs.Put**: To bind HTTP PUT requests to this method.
   2. **@Path(“{id}”)**: To map the requested URI **“/customers/{id}”.**
   3. The method takes two params. 
      1. int id: The value will be extracted from the URI using **@PathParam**
      2. **InputStream:** To read in the XML document.  
         Like we discussed earlier (createCustomer()), a method parameter not annotated with JAX-RS annotation is considered to represent the msg body of the request.

# Utility Methods

1. The final thing we have to discuss is the utility methods that we used in Service Methods in CustomerResource.java to transform Customer objects to and from XML.
   1. createCustomer()
   2. getCustomer()
   3. updateCustomer()
2. The **outputCustomer()** method takes a **Customer object** and writes it as **XML** to the response’s **OutputStream**.
3. A screen shot of a computer code

   Description automatically generated with low confidence
4. Next method is **readCustomer(InputStream)** responsible for reading XML text from an InputStream and creating a customer object.
   1. A picture containing text, font, screenshot

      Description automatically generatedA screen shot of a computer code

      Description automatically generated with low confidence
   2. The JDK has XML parser, so we don’t need to write ourselves or download a 3rd-party library to do it.
   3. The readCustomer(Inputstream) starts off by parsing the InputStream and creating a java object **model** that represents the **XML Document**.  
      The rest of the code in readCustomer(Inputstream), moves the data from the XML Model to Java customer object.
5. **NOTE**: In real system, we would not manually output XML or write all this boilerplate code to read in an XML document and convert it to a **business object**. But we don’t want to get off the learning JAX-RS basics by introducing another API.  
   In Chapter 6, we will learn how to use **JAXB** to map your customer object to XML and have JAX-RS automatically transform your HTTP message body to and from XML.

# JAX-RS And Java Interfaces

1. A picture containing text, screenshot, font, number

   Description automatically generated
2. Let’s transform Customer Resource Example into something that is **interface based**:
3. A picture containing text, screenshot, font

   Description automatically generated  
   A picture containing text, screenshot, font, line

   Description automatically generated
4. Let’s see the implementation for the above interface.  
   A screenshot of a computer program

   Description automatically generated with low confidence
5. For example, we want to **enforce a specific character set for POST XML**:  
   A picture containing text, screenshot, font, line

   Description automatically generated  
   The whole point of using interface to apply the JAX-RS metadata is to isolate the information and define it in one place.  
   If your annotations are scattered about b/w interface and implementation then our code will become a lot harder to read and understand.

# Inheritance

1. The **JAX-RS Specification** also allows you to define interface and class hierarchies if we desire so.
2. **For Example,** let’s say we want to make our readCustomer() and outputCustomer() methods abstract so that different implementations could transfer XML how they want.
3. A screenshot of a computer code

   Description automatically generated with low confidence
4. Then we can extend this abstract class and define readCustomer() outputCustomer()  
     
   A picture containing text, screenshot, font, line

   Description automatically generated  
   The only caveat with this approach is that the concrete subclass must annotate itself with @Path annotation to identify it as a service class to the **JAX-RS Provider** (Abstract doesn’t have @Path at class level but sub-class has it)

# Deploying Our Service

1. It is easier to deploy **JAX-RS** within a **Java-EE Certified Server** (e.g., **JBOSS**) or Standalone Server 3 Container (e.g., Tomcat).
2. Before we could do that, we need one simple class that extends **java.ws.rs.core.Application**.
3. This class will tell our app server which **JAX-RS Components** we want to register.
4. A picture containing text, screenshot, font, line

   Description automatically generated
   1. The **getClasses()** method returns a list of **JAX-RS Service classes** (and providers, but we’ll go to that in Chapter 6)
      1. Any JAX-RS Service returned by this method will follow **per-request model**.
      2. When the JAX-RS vendor Implementation determines that an HTTP request needs to be delivered to a method of one of these classes, an instance of that class will be created for the duration of the request and thrown away.  
         It means you’re delegating the creation of these objects to the **JAX-RS runtime**.
      3. For our Customer Service Database Example, we don’t have any per-request services, so **ShoppingApplication.getClasses**() will return empty set (See the next slide)
   2. The **getSingletons()** method returns a list of **JAX-RS Service Objects** (and providers, we will discuss in Chapter 6).  
      As these objects are singleton so application programmers are responsible for creating and initializing these objects.
      1. **ShoppingApplication.getSingletons()** returns a set of size one (See the next slide) containing CustomerResource instance.

These two methods tell the JAX-RS vendors which services we want deployed.  
**Example**:  
A screenshot of a computer program

Description automatically generated with low confidence  
**@ApplicationPath** defines the **relative Base URL** path for all the **JAX-RS services** in the deployment.

So, all our **JAX-RS RESTful services** will be pre-fixed with **the /services path**.

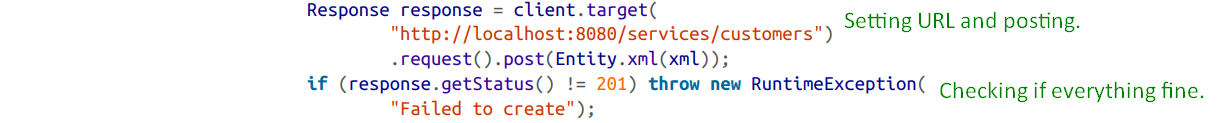
1. In **Java EE and Standalone servlet deployments, JAX-RS classes** must be deployed within the **Application Server’s Servlet Container** as a WAR (**W**eb **AR**chive).  
   Think of **Servlet Container** as your **Application Server**’s **Web Server**.  
   A **WAR** is a JAR File, in addition to Java Class Files, also contains other Java Libraries along with the Dynamic (like JSPs) and static content (Like HTML files or images) that we want to publish on our website.  
   We need to place our Java Classes within this archive so that our application server can deploy them.  
   Here is what the structure of a WAR file looks like:  
   A picture containing text, screenshot, font, algebra

   Description automatically generated
2. Because this example deploys within a Java EE Application server or standalone Servlet 3.x container, all we need is an empty web.xml file (as the JAX-RS Vendor Implementation will be provided by the Server and we don’t want to configure any index.html).  
   The server will detect that an **Application class** is within your WAR and automatically deploy it. Our app is not ready to use!

# Writing A Client

1. We can use **JAX-RS 2.0 Client API** to interact with a remote RESTful Service like we just created.
2. The **Client Interface** is responsible for managing client **HTTP Connections**.  
   We will discuss more about Client in Chapter 8.
3. Let’s take a look how to create a customer by invoking the remote service defined earlier in this capter.
4. Example:  
   A picture containing text, screenshot, font

   Description automatically generated  
   A screen shot of a computer code

   Description automatically generated with low confidenceA picture containing text, screenshot, font

   Description automatically generated  
   A screen shot of a computer code

   Description automatically generated with low confidenceA close-up of a text

   Description automatically generated with low confidence

# Wrapping UP

1. In this chapter, we discussed how to implement a **simple Customer Database as a JAX-RS Service.**
2. A picture containing text, screenshot, font, line

   Description automatically generated