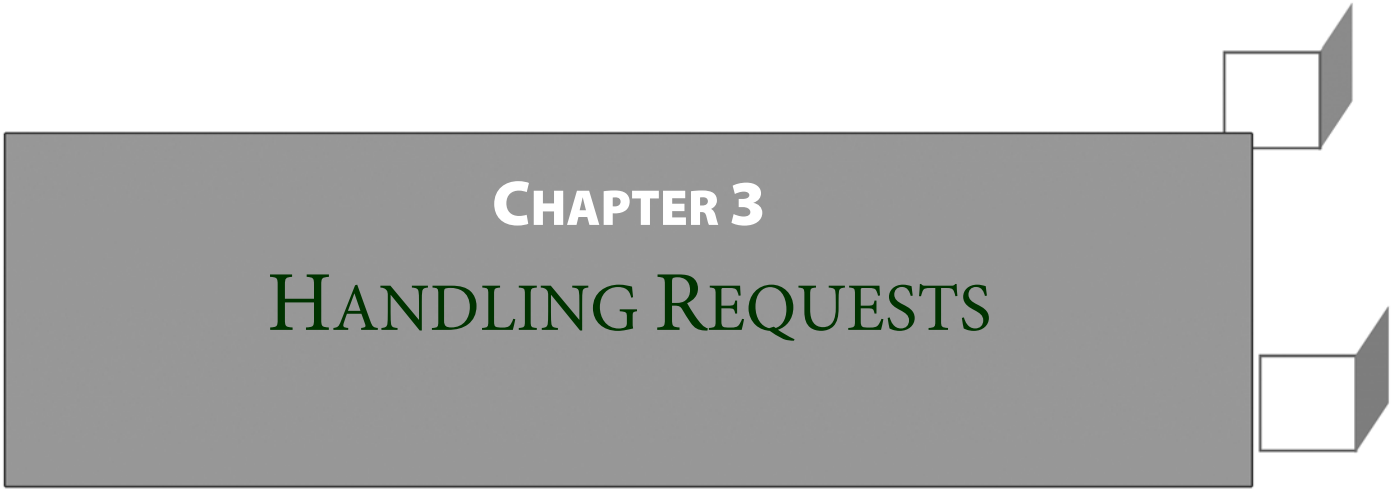
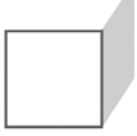
Chapter 3

Handling Requests





OBJECTIVES

After completing "Handling Requests," you will be able to:

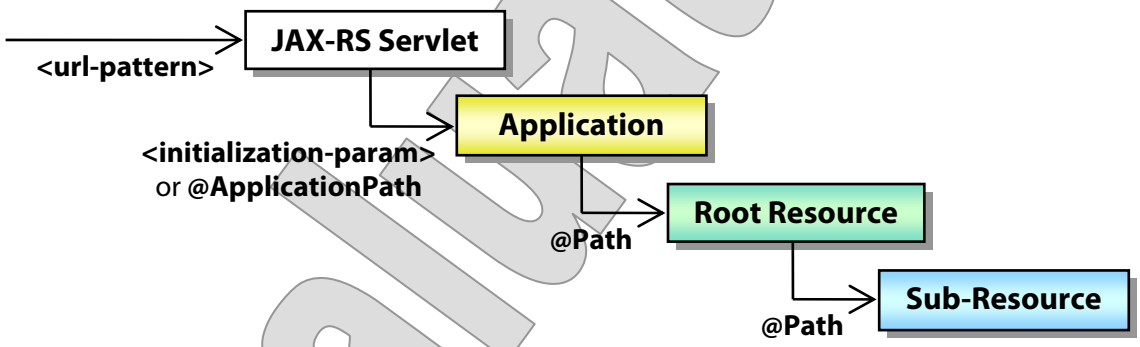
- **Explain how URLs are ultimately dispatched to Java methods based on path values for each of the application class, root resource class, and sub-resource method.**
- **Control precisely which Java method on which Java class is invoked for a given HTTP request, based on**
  - Request method
  - Path pattern
- **Use sub-resource locators to further partition the URL space for your web service.**
- **Bind various types of request content to Java method parameters, using JAX-RS annotations:**
  - Query parameters
  - Form parameters
  - Path parameters
  - Headers, cookies, and matrix parameters
- **Build adapters and converters to allow binding of parameters to non-standard Java types.**

Connecting HTTP to Java

- **At bottom, JAX-RS is a mechanism by which a Java method can act as an HTTP request handler.**
- **What's that you say? We already have such a mechanism, and it's called "servlets?"**
- **True, as far as it goes: doGet and doPost are Java methods that handle HTTP requests.**
- **The key differences that make JAX-RS valuable are these:**
  - It provides a more **sophisticated mapping** of URLs to requests, taking different HTTP methods into account and also allowing for a **hierarchy** of classes and methods.
  - The methods themselves offer natural **Java semantics**, with parameter and return types that fit the actual business purpose, rather than the HTTP-oriented signatures of servlets methods.

How It Works

- JAX-RS relies on a **front-controller servlet** to handle requests: this is a vendor-specific class with standard responsibilities.
- When a request arrives, the servlet initiates a **cascading dispatch** heuristic by which the proper Java method is invoked.
- So a JAX-RS developer organize resources at several scopes:



  - The web application can have one or more JAX-RS **applications**
  - An application publishes objects or classes as **root resources**
  - A root resource offers **sub-resource methods**
- Each of these entities can define a distinct path or pattern, and JAX-RS uses those paths and patterns to decide who gets the call.
- It is possible then for the ultimate URL to be of the form

```
/ContextRoot/JAX-RSRoot/AppPath/ResPath/SubResPath
```

  - Then there are many wrinkles on this basic formula.
- Finally, classes can be sub-resources as well; we'll return to that possibility later in the chapter.

Resources

- **A root resource is any class that is annotated with @Path.**
- **So, the application publishes the resource – but doesn't bind it to any URL pattern.**
- **It's the resource that announces this binding, for itself.**
- **A root resource's @Path may be significant, or not.**
  - Any non-empty **value** of the annotation will contribute to the **matching template** for incoming URLs.
  - An empty value is fine, and can make for shorter request URL patterns overall.
  - The path must be **unique** within the scope of the publishing application – and so no two root resources can have empty paths, or the same significant paths.
- **Then, methods may just be annotated for specific HTTP methods, as in @GET, @PUT, etc. These are called resource methods.**
- **A root resource class may also identify sub-resources. These can be ...**
  - **Methods**, usually also identified with **@Path** annotations and always with annotations identifying **HTTP methods**
  - **Classes**, which are identified by special methods known as **sub-resource locators** and which are annotated with **@Path** but not with HTTP methods

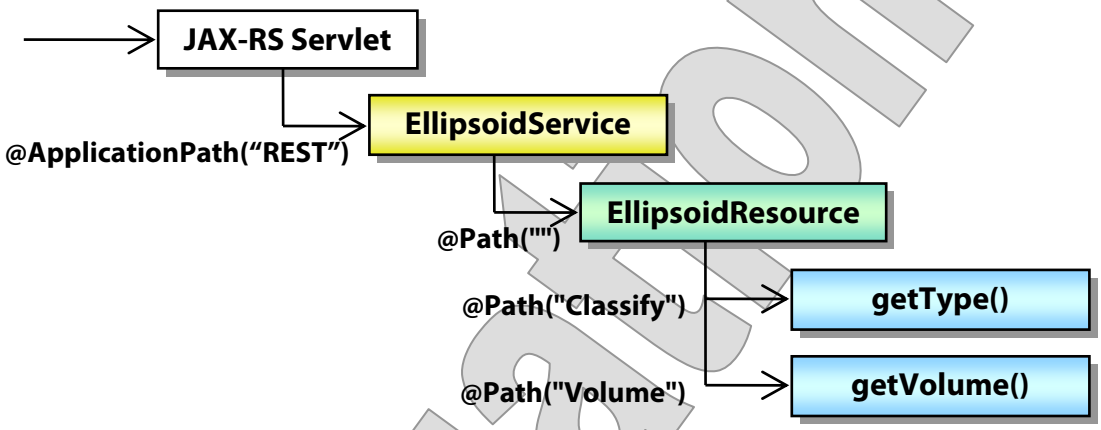
Sub-Resources

- **A method on a resource class that is annotated with a specific @Path, but not annotated to respond to a specific HTTP request method, is understood to be a sub-resource locator.**
- **The object it returns is a sub-resource, available at the given path and capable of matching to incoming requests from there.**
- **Sub-resource classes are not annotated with @Path.**
  - If they were, they would be mis-interpreted as **root resources**, and would be addressable directly under the web context root.
  - The paths at which they are available are dictated by the sub-resource locator methods that instantiate them.
- **But their methods can be annotated just the way root-resource methods can, with HTTP method and path annotations.**
- **A sub-resource class can have sub-resource locators – potentially making for a chain of sub-resources, of any desired length.**

The Ellipsoid Service

EXAMPLE

- The results of last chapter's lab, in **Ellipsoid_Step3**, include an application, root resource, and sub-resource method:



- The resulting request URL is composed of these four values – but in this case, two of them are empty.
  - This is common enough: we have the tools to create separate URLs at all these different scopes, but often it's nice to **collapse** some of them so that the URLs are simpler.
  - So in this case the request will come to the following URL – given that the web context root URL will be "/Ellipsoid":

```
/Ellipsoid/REST/Classification
/Ellipsoid/REST/Volume
```

Understanding JAX-RS Request URLs

- **Now, let's consider some of those "wrinkles" on the basic path-construction system for JAX-RS.**
- **We saw just now that it's okay for some path components to be, essentially, null. More generally:**
  - **@ApplicationPath** is not required, and it's typical to declare either this or a **`<url-mapping>`**, but not both.
  - **@Path** is required for root-resource classes and sub-resource methods – but it can have the empty string as a value.
  - Missing and empty-string path tokens will essentially **disappear** from the URL – while any non-empty content will be included along with a leading **/** character.
  - For empty-string paths to work, the thing being annotated can't have any **direct siblings**; for example you can't have two sub-resource methods and either of them annotated **@Path("")**.
- **@Path values for sub-resource methods can also be patterns that include path parameters.**
  - This allows part of the path to identify a value, such as the ID of a requested object.

```
/HR/Employees/5302/History
```

  - Different numbers in this sort of URL don't trigger dispatching to different methods, but instead become arguments to some parameter defined on a single method.

Uniqueness and Precedence

- **Each path must be unique in its scope:**
  - **Applications** must be mapped uniquely in the web application
  - **Root resource paths** must be unique for their application
  - **Sub-resource paths** must be unique under their parent resource
- **It is usually a good idea for the application path to be significant: map either to a non-empty URL pattern or @ApplicationPath.**
  - An empty path can work, but it claims all possible URLs for the web application, so it's appropriate only if the entire WAR is dedicated to one JAX-RS application, and always will be.
  - Want to add HTML pages, JSPs, or other servlets later? You'll have to refactor your paths, or maybe do something overly complicated with servlet filters.
- **For a really fun read, check out the exact request-matching heuristic: see the JAX-RS specification, section 3.7.2.**
- **But for the most part these rules resolve to an intuitive system.**
  - **Application** is determined first, then **root resource**.
  - JAX-RS then determines the correct **resource**: this could be the root resource, a sub-resource class, etc.
  - It then finds a **method** that matches – based on path as well as HTTP method and content types in and out.
- **One important rule to keep in mind is that a literal path token always takes precedence over a path parameter.**
  - We'll see this by example in a moment.

The HTTP Method Annotations

- A set of JAX-RS annotations exists just to identify HTTP methods:

```
@DELETE
@GET
@HEAD
@OPTIONS
@POST
@PUT
```

- In fact each of these, as an annotation type, is itself annotated with **@HttpMethod**.

```
@Target(value=METHOD)
@Retention(value=RUNTIME)
@HttpMethod(value="POST")
public @interface POST
```

- This is what formally sets these annotations apart, so that they can be used to distinguish sub-resource methods.
- A sub-resource method may have a **@Path** annotation, but must have exactly one HTTP-method annotation.
- There is also built-in support for two HTTP methods:
  - A **HEAD** request will be dispatched to a **@HEAD** method or, if none is found to a **@GET** method, discarding any response body.
  - For an **OPTIONS** request, if there is no **@OPTIONS** method, the JAX-RS implementation must generate a response based on the available metadata for the request URL.
  - ... and that's all the specification says about that! and different providers do different things here.

Sorting Out Paths

EXAMPLE

- In **Basic_Step1** there is an application that publishes several root resource classes.
- Each of these is designed to give some feature of JAX-RS a pretty thorough run around the block.
  - We've seen the **DB** class that implements a CRUD API over a map from letter keys to numeric values.
- The application is defined in **src/cc/jaxrs/BasicServices.java**:

```
@ApplicationPath("")
public class BasicServices
  extends Application
{
  @Override
  public Set<Class<?>> getClasses ()
  {
    Set<Class<?>> result = new HashSet<Class<?>> ();
    result.add (DB.class);
    result.add (Hello.class);
    result.add (Home.class);
    return result;
  }

  @Override
  public Set<Object> getSingletons ()
  {
    return Collections.singleton
      ((Object) new CarLot ());
  }
}
```

Sorting Out Paths

EXAMPLE

- The last of these root resources mixes various paths for different purposes: see **src/cc/jaxrs/CarLot.java**:

```
@Path("Car")
public class CarLot
{
  @GET
  @Path("Inventory")
  public String getInventory () { ... }

  @Path("{VIN}")
  public Car getCar (...) { ... }

  @GET
  @Path("{VIN}/year")
  public String getYear (...) { ... }

  @DELETE
  @Path("{VIN}/Delete")
  public void deleteCar (...) { ... }
}
```

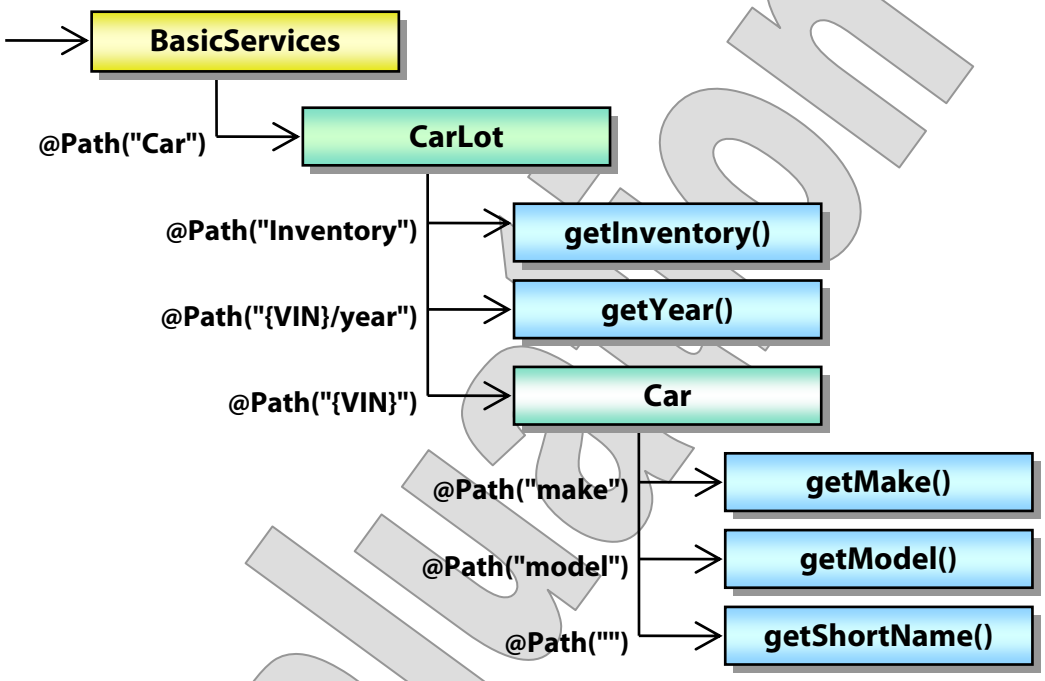
- So **cc.jaxrs.Car** is a sub-resource class, and it has its own sub-resource methods, returning individual values:

```
  @GET @Path("make")
  public String getMake ()

  @GET @Path("model")
  public String getModel ()

  public int getYear ()
  ...
  @GET @Path("")
  public String getShortName ()
```

Sorting Out Paths

EXAMPLE

- So there are some interesting pattern-matching cases here:



  - All relevant URLs start with **`/Basic/Car`**.
  - Under that, **`/Inventory`** goes to the sub-resource method **getInventory**, because this more specific pattern trumps the more general one with the **{VIN}** path parameter.
  - **`/ED9876`**, **`/PV9228`**, and indeed any other next token will go through the sub-resource locator method **getCar**, and on from there to a sub-resource method on the **Car** class.
  - This implies that a car with a VIN of "Inventory" could not be addressed; we'll assume that's not a realistic use case.
  - If the URL then ends with the VIN, **getShortName** is invoked.
  - If it continues with **`/make`**, **`/model`**, or one of a few other annotated paths, **getMake**, **getModel**, etc. will be invoked.

Sorting Out Paths

EXAMPLE

- So the following URL breaks down into two parts:

```
/Basic/Car/ED9876/make
```

  - **BasicServices** defines an empty application path.
  - `/Car` identifies a root resource, **CarLot**, and `/ED9876` a sub-resource class, **Car**.
  - `/make` identifies the **getMake** method.
- But, finally, note that a VIN followed by `/year` will be sent to **CarLot.getYear** – not **Car.getYear**.

```
/Basic/Car/ED9876/year
```

  - Now, `/ED9876/year` identifies **CarLot.getYear** – taking precedence over the initial match to the **getCar** locator method.
  - The **Car** class never gets a chance to match anything.
  - In this case we do this because **Car.getYear** returns an **int**, and this is not a viable return type for a JAX-RS sub-resource method.
  - We could rework that method, of course, but in all other respects Car is an ordinary JavaBean; no changes to its pre-existing semantics have been made.
  - So this is one of a few techniques to leave an existing class unmolested, while wrapping its logic for RESTful service – or, to re-use an existing sub-resource class, while overriding some of its behavior by intercepting certain requests that it might otherwise handle.

Sorting Out Paths

EXAMPLE

- You can build and test this application by running the **HTTP/Paths.txt** script in HTTPPad.
- A few requests and responses are shown below:

```
GET /Basic/Car/Inventory HTTP/1.1

2004 Toyota Prius (Black)
2004 Subaru Outback (Green)
2003 Ford Taurus (Gold)
2004 Saab 9000 (Silver)
2003 Saturn Ion (Plum)
1977 AMC Pacer (Blue)
1974 Ford Pinto (Dust)
1978 Renault Le Car (Yellow)
1991 Geo Metro (Midnight)


GET /Basic/Car/PV9228/ HTTP/1.1

2004 Subaru Outback (Green)


GET /Basic/Car/PV9228/make HTTP/1.1

Subaru


GET /Basic/Car/PV9228/year HTTP/1.1

2004
```

Sorting Out Methods

EXAMPLE

- **Let's look again at src/cc/jaxrs/DB.java, which exposes four methods according to a common CRUD pattern:**
  - The class sets its path as a root resource:

```
@Path("Number")
public class DB
{
```

  - Then, see that the methods all have the same **@Path**. Of course they all share the base path, and so are distinguished solely by their HTTP-method annotations:

```
  @GET
  @Path("{key}")
  public String get (@PathParam("key") String key)
  ...
  @POST
  @Path("{key}")
  public String post
    (@PathParam("key") String key, String value)
  ...
  @PUT
  @Path("{key}")
  public String put
    (@PathParam("key") String key, String value)
  ...
  @DELETE
  @Path("{key}")
  public String delete
    (@PathParam("key") String key)
  ...
}
```

  - The path value consists only of a **path parameter** – a technique we'll consider later in this chapter.

Sorting Out Methods

EXAMPLE

- We can test this more completely than we did earlier, using the script **HTTP/Methods.txt**.
- Below is a condensed reproduction of the requests and responses:

```
GET /Basic/Number/F HTTP/1.1

HTTP/1.1 200 OK
Content-Type: text/plain

6


PUT /Basic/Number/F HTTP/1.1
Content-Type: text/plain

100

HTTP/1.1 204 No Content


GET /Basic/Number/F HTTP/1.1

HTTP/1.1 200 OK
Content-Type: text/plain

The value of F is 100

... (PUT the value back, GET it again) ...
```

Sorting Out Methods

EXAMPLE

```
POST /Basic/Number/K HTTP/1.1
Content-Type: text/plain

11

HTTP/1.1 201 Created


GET /Basic/Number/K HTTP/1.1

HTTP/1.1 200 OK
Content-Type: text/plain

11


DELETE /Basic/Number/K HTTP/1.1

HTTP/1.1 204 No Content


GET /Basic/Number/K HTTP/1.1

HTTP/1.1 500 Internal Server Error
Content-Type: text/html;charset=utf-8

(and it gets ugly from here ...)
```

- See **HTTP/Methods_Results.txt** for the whole, annotated conversation.
- Error handling, and especially that last response, is going to need some attention. We'll tighten this up in an exercise later in the course.

Annotation Inheritance

- **A subclass of a JAX-RS resource class can inherit the method annotations on that base class.**
- **It is also possible to define an interface with JAX-RS methods, and then implement that interface on a resource class, thus inheriting the method annotations.**
- **In fact you have three options for any JAX-RS-annotated method on your base type:**
  - Inherit the **method** and its **annotations**
  - Override the method, but don't annotate it – this will allow you to inherit the **annotations**
  - Override the method and annotate it, in any way – this disables annotation inheritance completely and you inherit **nothing**
- **So, in a derived class, you can work method-by-method, but not annotation-by-annotation: it's all of them or none.**
- **It's a little early to get into examples of annotation inheritance.**
- **It's most useful for larger applications for which there is value in defining a pattern of methods for all resources, and then specializing that pattern for certain resource types.**
- **We will see an example of annotation inheritance in a later chapter.**

@Consumes and @Produces Annotations

- **HTTP requests use headers to declare a their content types and to announce the types that are acceptable in a possible response.**

```
Content-type: text/xml
Accept: image/gif;image/jpg
```

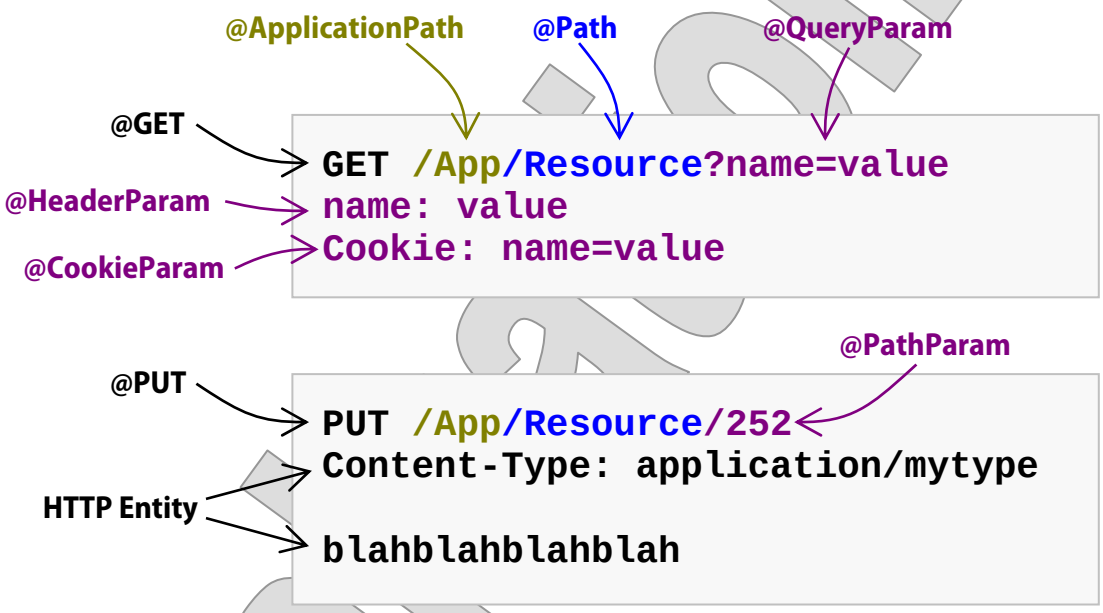
- **JAX-RS uses two annotations to fix what it calls the media types for a given method:**

```
@Consumes
@Produces
```

- **Each of these annotations, when present, further qualifies candidate requests as possible matches to the annotated method.**
  - If a method that would otherwise match an incoming request – that is, by HTTP method and URL – is annotated with **@Consumes**, and the request's content type is not found in the list of "consumable" types, the method is not considered suitable and will not be invoked.
  - Similarly, if annotated with **@Produces**, and the request has an **Accept** header, and there is no intersection between the "producible" types and the "acceptable" types, that's a show-stopper.
- **@Produces can also inform JAX-RS of the appropriate MIME type(s) for the response, once it has been chosen to handle a request.**
- **We'll consider these annotations in greater depth in later chapters, when we start to focus on HTTP entities.**

Decomposing HTTP Requests

- JAX-RS breaks an incoming HTTP request into its components, at a fairly fine grain.



- Then, through a combination of annotations and Java types, it gives you visibility to those components.
  - **Matching annotations** are those that help to determine what Java method is invoked to handle a given request: we've seen most of these, including **@GET**, **@PUT**, **@POST**, **@DELETE**, **@ApplicationPath**, and **@Path**.
  - **Parameter annotations** then determine what parts of the request are bound to method parameters. Shown above are **@QueryParam**, **@PathParam**, **@HeaderParam**, and **@CookieParam**, and there are a few others.
- The request **entity** may be bound to a method parameter, as well – but not by annotation.

Binding Request Content

- **Once an HTTP request is bound to a handling method, JAX-RS classifies its grammatical parts as**
  - **Parameters**, including parameters in the query string, POSTed form parameters, but also headers and cookies, matrix parameters, and even parts of the URL
  - The request **entity** – which is a sort of fancy name for the message body, but taken as content of some defined type and not just as a stream of bytes
- **There is one system for absorbing all those different types of parameters, using some combination of**
  - **Fields**
  - **Constructor parameters**
  - Request-handling **method parameters**
- **There is another system entirely for handling entities, and we look at that in several later chapters.**
  - The entity system also manages **response entities**.

Annotating Fields and Parameters

- **A JAX-RS service method can define any number of parameters.**
- **Each of those parameters must be annotated to inform JAX-RS as to how to provide a value for it – with one exception.**
  - Use **parameter annotations** to absorb parts of the HTTP request.
  - Use **@Context**-annotated parameters of certain types to accept contextual information such as servlet context or security context.
  - There can be at most one parameter not so annotated. If present, this recognized as an **entity parameter**.
- **The rules for fields and constructors are similar.**
  - Any field can be annotated as above – and if not annotated is ignored by JAX-RS.
  - A constructor must have all annotated parameters; entity parameters are allowed only in service methods.
- **Note however that only method parameters are workable for singleton-scope resources, since they are set up in advance of any request being received.**
- **Sub-resources may enjoy some or all of these options, depending on how and when they are created.**
  - If created during request processing, using **ResourceContext.getResource**, they can use fields and constructor parameters.
  - If instantiated first and then passed to **initResource**, they can have injectable fields, but not constructor parameters.

The Parameter Annotations

- Parameters are all basically name-value pairs of some sort.
- Each type can be absorbed by a field or parameter, using annotations as shown below, along with examples of request content that they would absorb correctly:

```
public void foo (@QueryParam("x") int numberX)
GET /MyService/URL?x=56 HTTP/1.1


public void foo (@FormParam("x") int numberX)
POST /MyService/URL HTTP/1.1
Content-type: application/x-www-form-urlencoded

x=56


public void foo (@MatrixParam("x") int numberX)
GET /MyService/URL;x=56 HTTP/1.1


@Path("URLPattern/{x}")
public void foo (@PathParam("x") int numberX)
GET /MyService/URLPattern/56 HTTP/1.1


public void foo (@HeaderParam("x") int numberX)
GET /MyService/URL HTTP/1.1
x: 56


public void foo (@CookieParam("x") int numberX)
GET /MyService/URL HTTP/1.1
Cookie: x=56
```

Simple Parameter Types

- **Fields and parameters with parameter annotations may be of the following types:**
  - Any Java **primitive**
  - Any Java class for which there is a constructor that takes a single parameter of type **String** – note that this includes **String** itself
  - Any Java class for which there is a static method **valueOf** or **fromString** that takes a single parameter of type **String** and returns an instance of the class – note that this includes the **boxing types** (**Integer**, **Long**, etc.) and any **enumerated type**
  - A **List**, **Set**, or **SortedSet** of some other supported type
  - Any type for which a **parameter converter** has been registered
- **Parsing of input content is strict.**
  - For example, floating-point content set where an integer is expected will fail to be parsed – not simply be trimmed to the integral part of the number, but rejected whole.
  - Some JAX-RS implementations actually treat this as a failure to select the method, and send back an HTTP 404 rather than a 400.

Accepting Input

EXAMPLE

- Back to our old stomping grounds: **Basic_Step1** offers one more root resource class, which is dedicated to experiments with parameter types. See **src/cc/jaxrs/Hello.java**:
  - Several methods draw the same value in six different ways:

```
@Path("Hello")
public class Hello
{
  ...
  @GET @Path("Query")
  public String getQuery
    (@QueryParam("name") String name) { ... }

  @GET @Path("Path/{name}")
  public String getPath
    (@PathParam("name") String name) { ... }

  @GET @Path("Matrix")
  public String getMatrix
    (@MatrixParam("name") String name) { ... }

  @POST @Path("Form")
  public String getForm
    (@FormParam("name") String name) { ... }

  @GET @Path("Header")
  public String getHeader
    (@HeaderParam("name") String name) { ... }

  @GET @Path("Cookie")
  public String getCookie
    (@CookieParam("name") String name) { ... }

  ...
```

Accepting Input

EXAMPLE

- Typical input to these methods is found in **HTTP/Params.txt**:

```
GET /Basic/Hello/Query?name=query%20parameter
  HTTP/1.1
----
GET /Basic/Hello/Path/path%20parameter HTTP/1.1
----
GET /Basic/Hello/Matrix;name=matrix%20parameter
  HTTP/1.1
----
POST /Basic/Hello/Form HTTP/1.1
Content-type: application/x-www-form-urlencoded

name=form%20parameter
----
GET /Basic/Hello/Header HTTP/1.1
name: header parameter
----
GET /Basic/Hello/Cookie HTTP/1.1
Cookie: name=cookie parameter
```

- You can test this script now, and then we'll review other parts of the application. Results are:

```
Hello, query parameter!
Hello, path parameter!
Hello, matrix parameter!
Hello, form parameter!
Hello, header parameter!
Hello, cookie parameter!
```

Accepting Input

EXAMPLE

- All of the methods in the next part of **Hello.java** take query parameters – but they vary the parameter type:

```
@GET @Path("Query/byte")
public String getQuery
  (@QueryParam("number") byte number) { ... }

@GET @Path("Query/short")
public String getQuery
  (@QueryParam("number") short number) { ... }

@GET @Path("Query/int")
public String getQuery
  (@QueryParam("number") int number) { ... }

@GET @Path("Query/long")
public String getQuery
  (@QueryParam("number") long number) { ... }

@GET @Path("Query/float")
public String getQuery
  (@QueryParam("number") float number) { ... }

@GET @Path("Query/double")
public String getQuery
  (@QueryParam("number") double number) { ... }

@GET @Path("Query/boolean")
public String getQuery
  (@QueryParam("flag") boolean flag) { ... }

public enum Answer { YES, NO, MAYBE };
@GET @Path("Query/enum")
public String getQuery
  (@QueryParam("answer") Answer answer) { ... }
```

Accepting Input

EXAMPLE

- Find sample input in **HTTP/Types.txt**:

```
GET /Basic/REST/Query/byte?number=42 HTTP/1.1
----
GET /Basic/REST/Query/short?number=42 HTTP/1.1
----
GET /Basic/REST/Query/int?number=42 HTTP/1.1
----
GET /Basic/REST/Query/long?number=42 HTTP/1.1
----
GET /Basic/REST/Query/float?number=4.2 HTTP/1.1
----
GET /Basic/REST/Query/double?number=4.2 HTTP/1.1
----
GET /Basic/REST/Query/boolean?flag=true HTTP/1.1
----
GET /Basic/REST/Query/enum?answer=MAYBE HTTP/1.1
...
```

- Run this script to see the responses – along with those to some additional requests we've yet to discuss:

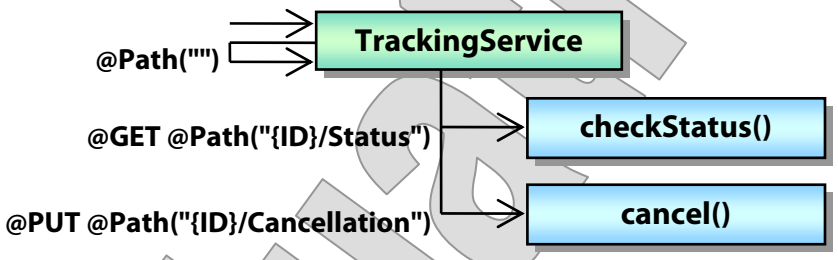
```
Hello, 42!
Hello, 42!
Hello, 42!
Hello, 42!
Hello, 4.2!
Hello, 4.2!
Hello, true!
Hello, MAYBE!
...
```

An Order-Tracking Service

LAB 3A

Suggested time: 30 minutes

In this lab you will begin the process of building up an order-tracking service for an online retailer. We'll start small, with just a couple of methods to check the status of existing orders and to cancel a given order:



Detailed instructions are found at the end of the chapter.

Accepting More Input

EXAMPLE

- **Finally the class works with String-convertible types, and with collections.**
  - We've already seen the **valueOf** approach work for an **enum**.
  - Now see the static inner class **AbstractPath**:

```
public static class AbstractPath
{
  private List<String> components =
    new ArrayList<String> ();

  public AbstractPath (String text)
  {
    for (String component : text.split ("/"))
      components.add (component);
  }

  public List<String> getComponents ()
  {
    return components;
  }
}
```

  - If we were to want to accept strings of the format "X/Y/Z" in path parameters, headers, etc., we could use this class directly as an annotated field or method parameter, because JAX-RS would use its **String**-based constructor automatically.

Accepting More Input

EXAMPLE

– Another service method does just that:

```
@Path("Query/Path")
@Produces("text/plain")
public String getQuery
  (@QueryParam("path") AbstractPath path)
{
  return greet
    ("" + path.getComponents ().get (0));
}
```

- The sample request shown next in the **Types.txt** script ...

```
GET /Basic/Hello/Query/Path
  ?path=Root/Child/Grandchild HTTP/1.1
```

– ... will give us a response that shows that the method was able to parse the path and give us back just the first component:

```
HTTP/1.1 200 OK
Content-Type: text/plain

Hello, Root!
```

Accepting More Input

EXAMPLE

- What about a class that has no **valueOf**, **fromString**, or String constructor – such as a **java.util.Date**?
  - We can adapt such a type by defining our own String-convertible class **DateAdapter**, which can be converted implicitly and then can give us the **Date** that it parsed:

```
public static class DateAdapter
{
  private Date date;

  public DateAdapter (String text)
  {
    try
    {
      date = new SimpleDateFormat
        ("M/d/yy").parse (text);
    }
    catch (ParseException ex) { ... }
  }

  public Date getDate ()
  {
    return date;
  }
}
```

Accepting More Input

EXAMPLE

- Now we can use this type as a JAX-RS parameter, and pull the value out of it in our service method:

```
@GET @Path("Query/Date")
public String getQuery
  (@QueryParam("date") DateAdapter adapter)
{
  return greet ("" + adapter.getDate ());
}
```

- So the next method shown in the script ...

```
GET /Basic/Hello/Query/Date?date=10/10/10 HTTP/1.1
```

  - ... produces a response with the **toString** representation of the parsed **java.util.Date**:

```
Hello, Sun Dec 6 00:00:00 EDT 2015!
```

Accepting More Input

EXAMPLE

- The last two methods receive collections of strings:

```
@GET @Path("Query/List")
public String getQuery
  (@QueryParam("names") List<String> names) {...}

@GET @Path("Query/Set")
public String getQuery
  (@QueryParam("names") Set<String> names) {...}
```

- The remainder of **HTTP/Types.txt** shows input to these two remaining service methods:

```
GET /@ROOT@/REST/Query/List
  ?names=Abbot&names=Abbot&names=Costello HTTP/1.1
----
GET /@ROOT@/REST/Query/Set
  ?names=Abbot&names=Abbot&names=Costello HTTP/1.1
```

- The remainder of the output from the test you just performed shows the response content for each:

```
...
Hello, 3 people!
Hello, 2 unique people!
```

The @DefaultValue Annotation

- **Any failure to parse an input will result in the parameter being given whatever is the default value for its type: `false` for `boolean`, zero for numbers, etc.**
- **But, any parameter can carry the `@DefaultValue` annotation.**
  - This lets you set your preferred default value.
  - This value will be used whenever the expected input is missing.

```
@HeaderParam("Mode")
@DefaultValue("Normal")
private String mode;

public void foo
  (@QueryParam("x") @DefaultValue("1") int x)
```

```
GET /MyService/URL?x=56 HTTP/1.1
```

- **Note that the value of this annotation is always expressed as a string, because it stands in for any missing information in the HTTP request, prior to parsing and type conversion.**

Defaults for Ellipsoid

DEMO

- **Let's put our Ellipsoid service back into a per-request mode, and then we'll set default values for the dimensions.**
  - Do your work in **Ellipsoid_Step3**.
  - The completed demo is found in **Ellipsoid_Step4**.

1. First, remove **src/cc/math/EllipsoidService.java** completely.
   - This class was publishing our resource as a singleton.
2. Move **web.xml** once again into **docroot/WEB-INF**.
   - Now the default application will find us based on our **@Path** annotation and will instantiate once per request.
3. In **src/cc/math/EllipsoidResource.java**, remove the lists of parameters from methods **getType** and **getVolume**.

```
@GET
@Path("Classification")
public String getType
(
   @QueryParam("a") double a,
   @QueryParam("b") double b,
   @QueryParam("c") double c
)
```

4. Instead, define three fields to accept the query parameters:

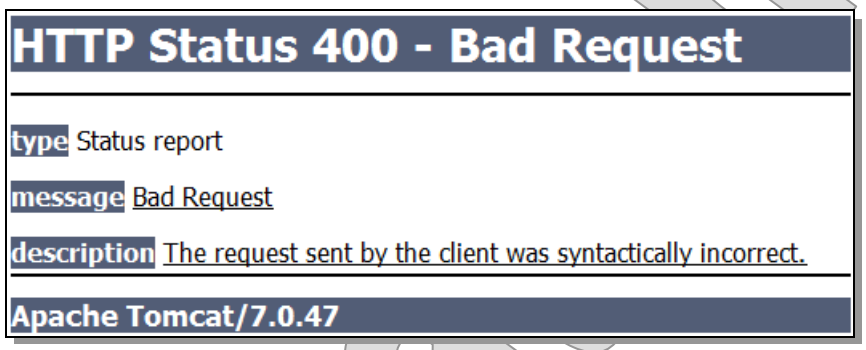
```
@QueryParam("a")
private double a;

@QueryParam("b")
private double b;

@QueryParam("c")
private double c;
```

Defaults for Ellipsoid

DEMO

5. Build and test the application as follows:

```
GET /Ellipsoid/REST/Classification HTTP/1.1
```



- Since no values were found, the JAX-RS implementation left the fields at their defaults of zero.
- This in turn triggered an **IllegalArgumentException**, thanks to some validation logic already in **Ellipsoid.java**:

```
public void setA (double newValue)
{
  if (newValue <= 0)
    throw new IllegalArgumentException
      ("All semi-axis lengths ...");

  a = newValue;
}
```

- Jersey converts this specific exception type to an HTTP 400 – a quirk which we'll discuss in a later chapter on error handling.

Defaults for Ellipsoid

DEMO

6. Set default values all around:

```
@QueryParam("a")
@DefaultValue("1.0")
private double a;

@QueryParam("b")
@DefaultValue("1.0")
private double b;

@QueryParam("c")
@DefaultValue("1.0")
private double c;
```

7. Build and test again, and see these in force:

```
GET /Ellipsoid/REST/Classification HTTP/1.1

HTTP/1.1 200 OK
Content-Type: text/plain

sphere


GET /Ellipsoid/REST/Classification?a=2.0 HTTP/1.1

HTTP/1.1 200 OK
Content-Type: text/plain

prolate spheroid
```

Parameter Converters

- JAX-RS 2.0 introduces another option, which is the **parameter converter**.
- A **parameter converter provider** can offer up any number of converters, each implementing **ParamConverter<T>** for a specific target type **T**.

```
interface ParamConverterProvider
{
  public <T> ParamConverter<T> getConverter
    (Class<T> cls, Type type,
      Annotation[] annotations)
}
```

- The converter itself implements parsing and printing methods:

```
interface ParamConverter<T>
{
  public T fromString (String text);
  public String toString (T object);
}
```

- So a converter can act as its own, dedicated provider; or a single provider can serve up a roster of available converters; or anything in-between.

A Date Converter

DEMO

- **Let's experiment with this feature by re-factoring our existing DateAdapter to be a DateConverter.**
  - Do your work in **Basic_Step1**.
  - The completed demo is found in **Basic_Step2**.

1. Create a new class **cc.jaxrs.Converters**.
2. Annotate it as a provider, and make it implement the required provider interface:

```
@Provider
public class Converters
  implements ParamConverterProvider
{
```

3. Create a static, inner class **DateConverter** that can convert the type **java.util.Date**:

```
public static class DateConverter
  implements ParamConverter<Date>
{
```

4. Define a date-formatter field that we'll use to do the parsing and printing. Note that you can copy some of this code from the existing **DateAdapter** found in **src/cc/jaxrs/Hello.java**.

```
  private SimpleDateFormat formatter =
    new SimpleDateFormat ("M/d/yy");
```

A Date Converter

DEMO

5. Implement the conversion methods to use the formatter:

```
    public Date fromString (String text)
    {
      try
      {
        return formatter.parse (text);
      }
      catch (ParseException ex)
      {
        throw new IllegalArgumentException
          ("Couldn't parse date: " + text, ex);
      }
    }

    public String toString (Date date)
    {
      return formatter.format (date);
    }
  }
}
```

   – The answer code goes a bit farther, defining a logger and logging the failure at warning level in the **catch** block.

6. In the outer class, initialize a field to a new instance of your converter:

```
private DateConverter dateConverter =
  new DateConverter ();
```

A Date Converter

DEMO

7. Finally, implement the converter-provider method to hand out your converter if the requested type is right:

```
@SuppressWarnings("unchecked")
public <T> ParamConverter<T> getConverter
  (Class<T> cls, Type type,
    Annotation[] annotations)
{
  if (cls == Date.class)
    return (ParamConverter<T>) dateConverter;

  return null;
}
```

8. Although your provider is annotated as a **@Provider**, and would be picked up by class scanning, this service uses an application subclass that explicitly publishes components. So, open **src/cc/jaxrs/BasicServices.java**, and add this class to those published in **getClasses**:

```
@Override
public Set<Class<?>> getClasses ()
{
  Set<Class<?>> result =
    new HashSet<Class<?>> ();
  result.add (DB.class);
  result.add (Hello.class);
  result.add (Home.class);
  result.add (Converters.class);
  return result;
}
```

A Date Converter

DEMO

9. In **src/cc/jaxrs/Hello.java**, remove the **DateAdapter** class.
10. In the overload of **getQuery** that takes a **DateAdapter**, switch over to accepting a **Date** directly, and then drop the call to **getValue** that you needed when using the adapter class:

```
@GET
@Path("Query/Date")
@Produces("text/plain")
public String getQuery
  (@QueryParam("date") DateAdapter date)
{
  return greet ("" + date.getValue ());
}
```

11. Deploy the application, and test as we did earlier in the chapter:

```
GET /Basic/Hello/Query/Date?date=10/10/10 HTTP/1.1

HTTP/1.1 200 OK
Content-Type: text/plain

Hello, Sun Oct 10 00:00:00 EDT 2010!
```

- **So, as we've seen, there are at least three options for parsing incoming parameters:**
  - Simply accept a **String** and do the parsing in your method.
  - Use an **adapter class** and then call its **getValue** or similar method.
  - Use a **parameter converter**.
- **Each of these offers progressively more centralized type-conversion logic, at higher cost; and so will be worth the trouble depending on how heavily the parameter type is used.**

Inputs and Outputs for Order Tracking

LAB 3B

Suggested time: 30 minutes

In this lab you will add features to your Tracking service: a query method that returns a list of order IDs based on order status, and a batch method that returns status for a list of ID values.

Detailed instructions are found at the end of the chapter.

SUMMARY

- **JAX-RS is in large part just a clever system for dispatching HTTP requests to Java methods.**
  - The system is **hierarchical**, with web applications, JAX-RS applications, root resources, and sub-resources, each of these listed things defining a scope for the next one.
  - It provides more natural semantics for methods, allowing the developer to bind HTTP values to Java method parameters and return types using source-code annotations.
- **This system fits the problem of RESTful service development better than other Java EE technology does.**
  - Servlets, JSP, and JSF are all geared to interactive applications.
  - JAX-WS is primarily dedicated to SOAP-oriented services, where there are many more assumptions about message content, MIME type, HTTP method, etc.
  - RESTful services often involve a variety of HTTP methods, URL encoding styles, and content types – and don't impose the assumption of symmetry over request and response, either, as SOAP-oriented services do.
  - JAX-RS is designed to support that whole panoply of options, largely by a combination of the URL mapping we've seen in this chapter and other schemes for binding request and response values to Java method signatures.

An Order-Tracking Service

LAB 3A

In this lab you will begin the process of building up an order-tracking service for an online retailer. We'll start small, with just a couple of methods to check the status of existing orders and to cancel a given order:

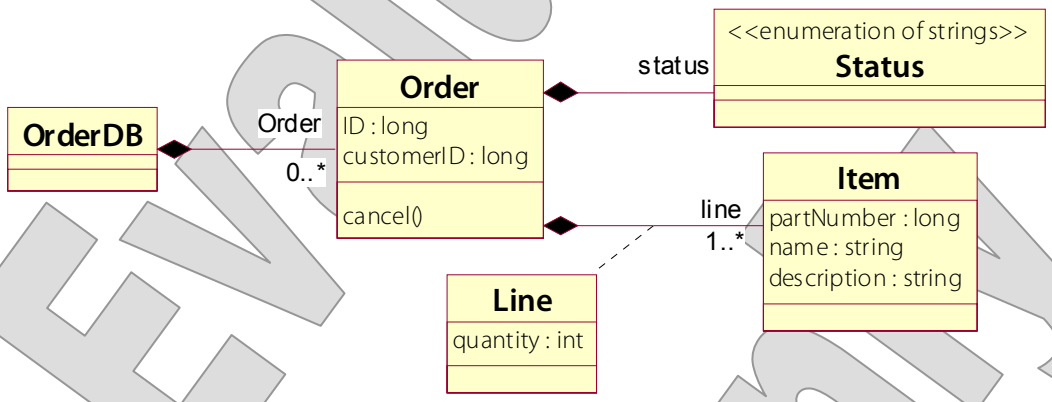
Lab project: **Tracking_Step1**

Answer project: **Tracking_Step2**

Files: **src/org/biz/TrackingApplication.java**
src/org/biz/TrackingService.java

Instructions:

1. Review **TrackingApplication.java**, which is annotated with an **@ApplicationPath** and publishes the **TrackingService** class as the only root resource.
2. Open **TrackingService.java** and see that there is a static field **DB**, of type **OrderDB**. A static initializer block builds up a small in-memory database of orders, according to the domain model found in package **org.biz.xml**:



As you may notice, these classes were generated by a JAXB compiler. For the moment we'll use them as POJOs (plain old Java objects, that is to say without regard for the JAXB annotations on and in them), but in later steps of the case study these annotations will help the application to load its data from a prepared XML file.

An Order-Tracking Service LAB 3A

3. Make the class a root resource by adding the **@Path** annotation, with an empty path string.
4. Add a helper method **getOrder** that loops through the list given by calling **DB.getOrder**, and returns the **Order** with an **ID** property that matches a value passed as a **long** parameter to the method.
5. Add a method **checkStatus** that takes a **long** parameter and returns a string.
6. Annotate the method to be a JAX-RS service method, such that it will be called when the application receives an HTTP GET request to a URL of the form "/{ID}/Status".
7. Annotate your method parameter to absorb the part of the URL that matches the {ID} path parameter.
8. Implement the method to call your helper method, passing the ID, and then call **getStatus** on that. Return the **toString** representation of that status value.
9. You should be able to test your service at this point. Deploy it on the server and test using HTTPPad as follows:

```
GET /Tracking/Orders/2/Status HTTP/1.1

HTTP/1.1 200 OK
Content-Type: text/plain

BACK_ORDERED
```

10. Add another service method **cancel**, taking a **long** and returning **void**.
11. Annotate it to handle an HTTP PUT to "/{ID}/Cancellation".
12. Annotate the method parameter to absorb the path parameter.
13. Implement the method to call **getOrder** and then call **setStatus** on the **Order** that it returns to you, passing **Status.CANCELED**.

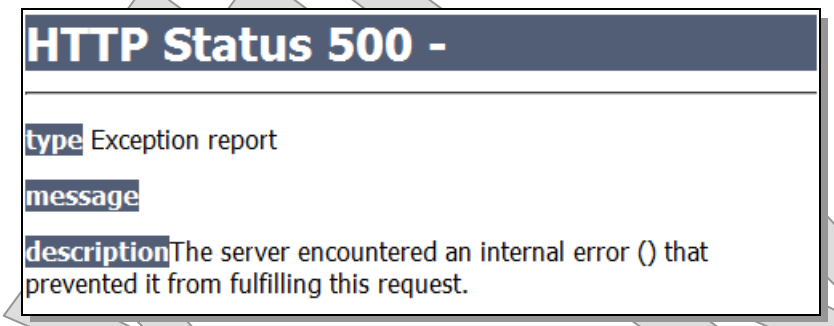
14. Deploy and test again as follows:

```
PUT /Tracking/Orders/2/Cancellation HTTP/1.1

HTTP/1.1 204 No Content


GET /Tracking/Orders/2/Status HTTP/1.1

HTTP/1.1 200 OK
Content-Type: text/plain

CANCELED
```

This is the answer code found in **Step2**.

Notice that an unknown ID is not handled all that cleanly:

```
GET /Tracking/Orders/399/Status HTTP/1.1
```



This is the JAX-RS implementation's best means of handling the **NullPointerException** that falls out of your service method when **getOrder** fails to find an order of the given ID. And that's something to fix up in a later exercise, once we've learned more about response production and exception handling.

Inputs and Outputs for Order Tracking

LAB 3B

In this lab you will add features to your Tracking service: a query method that returns a list of order IDs based on order status, and a batch method that returns status for a list of ID values.

Lab project: **Tracking_Step2**

Answer project: **Tracking_Step3**

Files: **src/org/biz/TrackingService.java**

Instructions:

1. Open **TrackingService.java**, and add a helper method **commaSeparatedList** that takes a **List<String>** and returns a comma-separated list of the values therein as a single string.
2. Add a service method **getOrdersByStatus** that takes a **Status** parameter and returns a **String**.
3. Annotate the method to respond to an HTTP GET. Don't add a **@Path** annotation: thus the method will represent the collection of orders itself. When a status value is provided it will be used to filter the results; when it is not provided this method will serve as a "get-all." (In later exercises you'll refactor this method to return the full order contents, instead of just the status flags.)
4. Annotate your method parameter to bind to an expected HTTP query parameter, also called "status".
5. Implement the method to read all the orders on **DB**, and to produce the IDs of orders whose status values match the requested status, in a comma-separated list.

Inputs and Outputs for Order Tracking LAB 3B

6. Build and test at this point, using HTTPPad to send the following request:

```
GET /Tracking/Orders?status=HOLD HTTP/1.1

HTTP/1.1 200 OK
Content-Type: text/plain

3
```

7. Add logic to your method to treat the request status of UNKNOWN as matching all orders.
8. Annotate the status parameter to have a default value of "UNKNOWN".
9. Test again, with the following request:

```
GET /Tracking/Orders HTTP/1.1

HTTP/1.1 200 OK
Content-Type: text/plain

1,2,3
```

10. Now add a second method **checkStatus** that takes a **List<Long>** as its parameter. Have it return a **String**.
11. Annotate the method to respond to an HTTP GET at "/Status".
12. Annotate the parameter to read all the values of a query parameter **ID**.
13. Implement the method to build up a response string by calling the original **checkStatus** method for each ID value in the list, and returning the comma-separated list of status values, using your helper method.
14. Build and test with the following request:

```
GET /Tracking/Orders/Status?ID=1&ID=3 HTTP/1.1

HTTP/1.1 200 OK
Content-Type: text/plain

PACKED,HOLD
```