Architecture of Enterprise Applications 17 RESTful Web Services

Haopeng Chen

***RE**liable, **IN**telligent and **S**calable Systems Group (**REINS**)*

Shanghai Jiao Tong University

Shanghai, China

http://reins.se.sjtu.edu.cn/~chenhp

e-mail: chen-hp@sjtu.edu.cn

RESTful Web Services

- RESTful Web Services
  - Why?
  - What?
  - How?
- RESTful Web Services in Java EE
  - JAX-RS
- Web Service or not?

Why

- SOAP-based Web Services
  - Coupling with the message format
  - Coupling with the encoding of WS
  - Parse and assemble SOAP
  - Need a WSDL to describe the details of WS
  - Need a proxy generated from WSDL

- It is a time-cost way to implement Web Service with SOAP
  - We should find a new way to implement WS

What

- REpresentational State Transfer
  - Representational:
    - All data are resources. Representation for client.
    - Each resource can have different representations
    - Each resource has its own unique identity(URI)
  - State:
    - It refers to state of client. Server is stateless.
    - The representation of resource is a state of client.
  - Transfer:
    - Client's representation will be transferred when client access different resources by different URI.
    - It means the states of client are also transferred.
    - That is Representation State Transfer

What

- REST is a kind of architecture, but not a specification
  - REST is a typical Client-Server architecture, but it is stateless server
  - All states are hold in the messages delivered between clients and server
  - Server only process the requirements of data, displaying is completely depended on clients
  - REST is idempotent which means server will return same results for same require. So the results can be cached on either clients or server

What

- In REST, all operations are preformed in unified way
  - Each resource has a unique identity
  - Process resource by representation
    - Client can not directly manipulate resources.
    - Client only can manipulate its representation, and send requires.
    - Server process requires and return response.
  - Any message between clients and server is self-described.
    - The context for processing a message is contained in the message itself.
  - Multimedia interaction system.
    - The content delivered between clients and server can be documents, pictures, audios, and videos
    - It is the base for resource to be rendered as different representations.

How to design REST

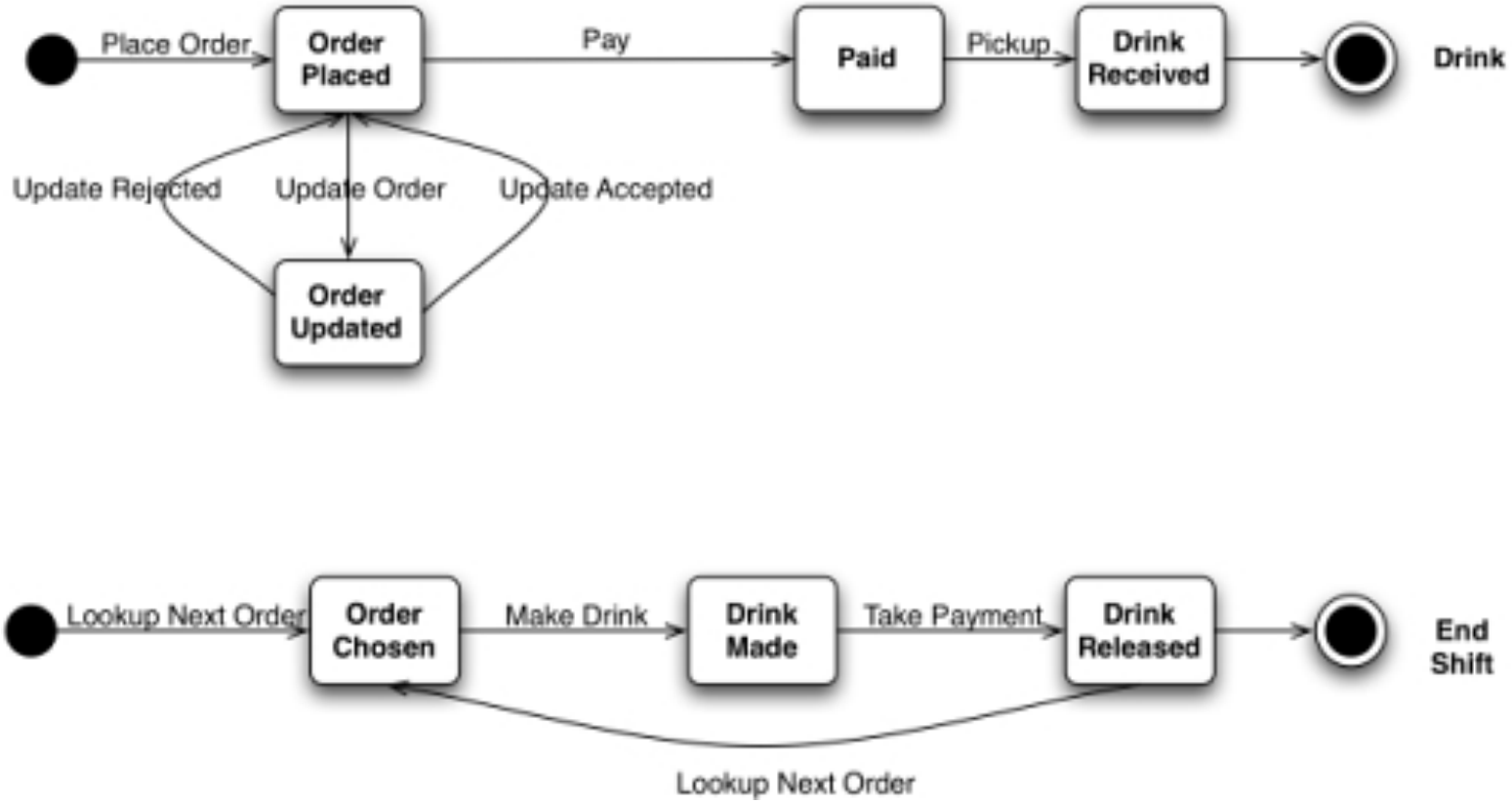
- Design rules
  - Anything on web is abstracted as resource
  - Any resource has a unique resource identifier
  - Access resource by generic connector interface
  - Any manipulation to resource doesn't change resource identifier
  - All operations are stateless

- Resources are not data, but the combination of data and representation
  - Same data with different representation will be abstracted as different resources.

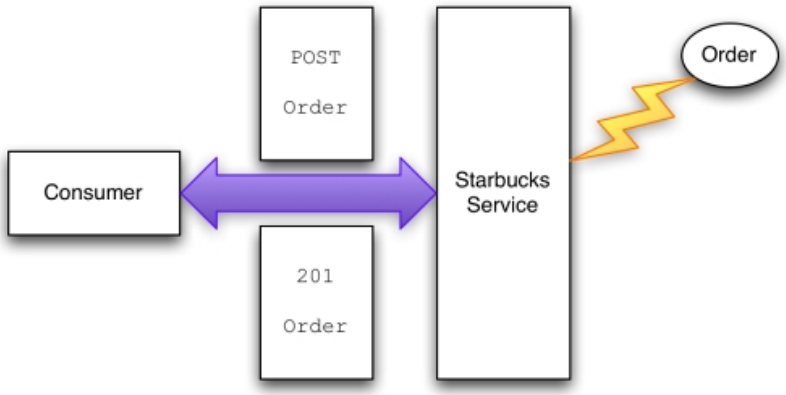
How to design REST

- CRUD
  - Atomic operations: Create, Read, Update, Delete
  - Composite them to build complex manipulate
- HTTP-based
  - GET-read
  - POST-create
  - PUT-update
  - DELETE-delete
- Design by URL
  - We just need to design suitable URLs which directly represent user interface
  - Developers just need to abstract resources according to URLs
  - URL without parameters is more convenient for user
  - Quite different from action-based design method, such as MVC
- Notice: it is very difficult to abstract anything on web as resource
  - Mix MVC and REST

How to Get a Cup of Coffee



How to Get a Cup of Coffee



Request:

```
POST /order HTTP1.1
Host: starbucks.example.org
Content-Type: application/xml
Content-Length: . . .

<order xmlns="http://starbucks.example.org/">
    <drink>latte</drink>
</order>
```

Response:

```
201 Created
Location: http://starbucks.example.org/order/1234
Content-Type: application/xml
Content-Length: . . .

<order xmlns="http://starbucks.example.org/">
    <drink>latte</drink>
    <cost>3.00</cost>
    <next xmlns="http://example.org/state-machine"
        rel="http://starbucks.example.org/payment"
        uri="http://starbucks.example.com/payment/order/1234"
        type="application/xml"/>
</order>
```

How to Get a Cup of Coffee

- Response code
  - 200 OK
  - 201 Created
  - 202 Accepted
  - 303 See Other
  - 400 Bad Request
  - 404 Not Found
  - 409 Conflict
  - 412 Precondition Failed
  - 417 Expectation Failed
  - 500 Internal Server Error

How to Get a Cup of Coffee

- Update order

Request	Response
`OPTIONS /order/1234 HTTP 1.1` `Host: starbucks.example.org`	`200 OK Allow: GET, PUT`

Request	Response

`PUT /order/1234 HTTP 1.1`
`Host: starbucks.example.com`
`Expect: 100-Continue`
	`100 Continue`

How to Get a Cup of Coffee

- Update order

Request:

```
PUT /order/1234 HTTP1.1
Host: starbucks.example.com
Content-Type: application/xml
Content-Length: . . .

<order xmlns="http://starbucks.example.org/">
    <additions>shot</additions>
</order>
```

Response:

```
200 OK
Location: http://starbucks.example.org/order/1234
Content-Type: application/xml
Content-Length: . . .

<order xmlns="http://starbucks.example.org/">
    <drink>latte</drink>
    <additions>shot</additions>
    <cost>4.00</cost>
    <next xmlns="http://example.org/state-machine"
        rel="http://starbucks.example.org/payment"
        uri="http://starbucks.example.com/payment/order/1234"
        type="application/xml"/>
</order>
```

How to Get a Cup of Coffee

- Update order

Request:

```
PUT /order/1234 HTTP1.1
Host: starbucks.example.com
Content-Type: application/xml
Content-Length: . . .

<order xmlns="http://starbucks.example.org/">
    <additions>shot</additions>
</order>
```

Response:

```
409 conflict
Location: http://starbucks.example.org/order/1234
Content-Type: application/xml
Content-Length: . . .

<order xmlns="http://starbucks.example.org/">
    <drink>latte</drink>
    <cost>4.00</cost>
    <next xmlns="http://example.org/state-machine"
        rel="http://starbucks.example.org/payment"
        uri="http://starbucks.example.com/payment/order/1234"
        type="application/xml"/>
</order>
```

How to Get a Cup of Coffee

- Payment

```
<next xmlns="http://example.org/state-machine"
      rel="http://starbucks.example.org/payment"
      uri="http://starbucks.example.com/payment/order/1234"
      type="application/xml"/>
```

Request	Response
`OPTIONS/payment/order/1234 HTTP 1.1` `Host: starbucks.example.com`	`Allow: GET, PUT`

How to Get a Cup of Coffee

- Payment

Request

```
PUT /payment/order/1234 HTTP 1.1
Host: starbucks.example.com
Content-Type: application/xml
Content-Length: ...
Authorization: Digest username="Jane Doe"
realm="starbucks.example.org"
nonce="..."
uri="payment/order/1234"
qop=auth
nc=00000001
cnonce="..."
reponse="..."
opaque="..."
123456789
07/07
John Citizen
4.00
```

Response

```
201 Created
Location:
https://starbucks.example.com/payment/order/1234
Content-Type: application/xml
Content-Length: ...
    123456789
    07/07
    John Citizen
    4.00
```

How to Get a Cup of Coffee

- Get a list of orders

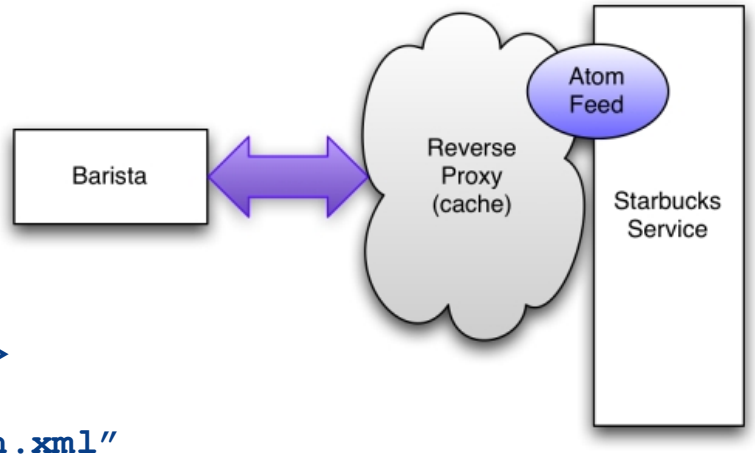
Response:

```
200 OK
Expires:Thu, 12Jun2008 17:20:33 GMT
Content-Type: application/xml
Content-Length: . . .

<?xml version="1.0" ?>
<feed xmlns="http://www.w3.org/2005/Atom">
    <title>Coffees to make</title>
    <link rel="alternate"
        uri="http://starbucks.example.com/orders"/>
    <updated>2014-05-16T08:18:43Z</update>
```

How to Get a Cup of Coffee

- Get a list of orders



Response:

```
<entry>
    <published>2014-05-16T08:18:43Z</title>
    <updated>2014-05-16T08:20:32Z</update>
    <link rel="alternate" type="application.xml"
        uri="http://starbucks.example.com/order/1234"/>
    <id>http://starbucks.example.com/order/1234</id>
    <content type="text+xml">
       <order xmlns="http://starbucks.example.com/">
          <drink>latte</drink>
          <additions>shot</additions>
          <cost>4.00</cost>
       </order>
       <link rel="edit"
          type="applicatioin/atom+xml"
          href="http://starbucks.example.com/order/1234"/>
       . . .
    </content>
</entry>
```

How to Get a Cup of Coffee

- Get a list of orders

Request

```
PUT /order/1234 HTTP 1.1
 Host: starbucks.example.com
 Content-Type: application/atom+xml
 Content-Length: ...

 <entry>
   . . .
   <content type="text+xml">
      <order xmlns="http://starbucks.example.com/">
         <drink>latte</drink>
         <additions>shot</additions>
         <cost>4.00</cost>
         <status>preparing</status>
      </order>
      . . .
   </content>
</entry>
```

How to Get a Cup of Coffee

- Check payment of orders

Request	Response

`GET /payment/order/1234 HTTP 1.1`
`Host: starbucks.example.org`
	`401 Unauthorized`
	`WWW-Authenticate: Digest`
	`realm="starbucks.example.com",`
	`qop="auth",`
	`nonce="ab656...",`
	`opaque="b6a9..."`

Request	Response

`GET /payment/order/1234 HTTP 1.1`
`Host: starbucks.example.org`
`Authorization: Digest`
`username="barista joe"`
`realm="starbucks.example.com"`
`nonce="..."`
`uri="payment/order/1234"`
`qop=auth`
`nc=00000001 c`
`nonce="..."`
`reponse="..."`
`opaque="..."`
	`200 OK`
	`Content-Type: application/xml`
	`Content-Length: ...`
	`123456789`
	`07/07`
	`John Citizen`
	`4.00`

How to Get a Cup of Coffee

- Delete a order

Request	Response
`DELETE /order/1234 HTTP 1.1` `Host: starbucks.example.org`	`200 OK`

RESTful Web Services in Java EE

- JAX-RS is a Java programming language API designed to make it easy to develop applications that use the REST architecture.

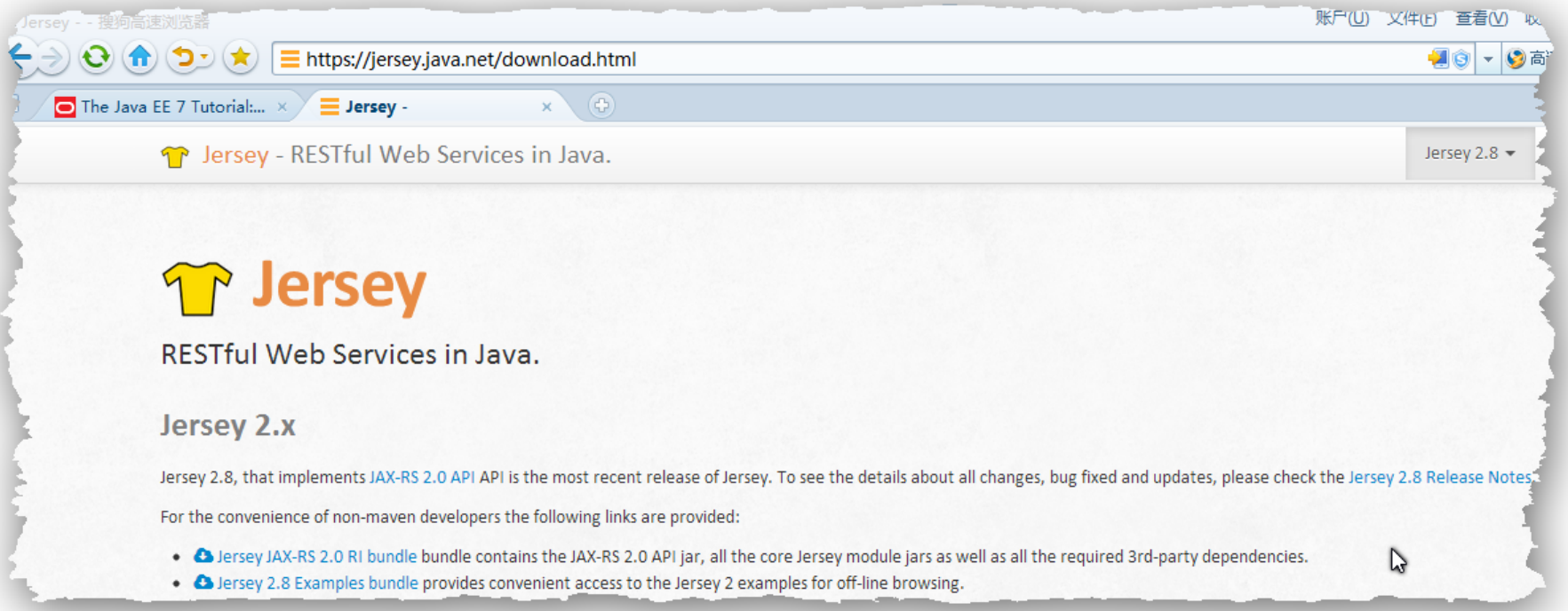
- @Path
  - The @Path annotation's value is a relative URI path indicating where the Java class will be hosted: for example, `/helloworld`.
- @GET @POST @PUT @DELETE @HEAD
  - The @GET @POST @PUT @DELETE @HEAD annotation is a request method designator and corresponds to the similarly named HTTP method.
- @PathParam @QueryParam
  - The @PathParam @QueryParam annotation is a type of parameter that you can extract for use in your resource class.
- @Consumes
  - The @Consumes annotation is used to specify the MIME media types of representations a resource can consume that were sent by the client.
- @Produces
  - The @Produces annotation is used to specify the MIME media types of representations a resource can produce and send back to the client
- @Provider
  - The @Provider annotation is used for anything that is of interest to the JAX-RS runtime, such as MessageBodyReader and MessageBodyWriter.
- @ApplicationPath
  - The @ApplicationPath annotation is used to define the URL mapping for the application.

Overview


```
package javaeetutorial.hello;

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
……
/** * Root resource (exposed at "helloworld" path) */
@Path("helloworld")
public class HelloWorld {
 @Context
 private UriInfo context;
 /** Creates a new instance of HelloWorld */
 public HelloWorld() { }
 /** * Retrieves representation of an instance of helloWorld.HelloWorld
  * @return an instance of java.lang.String */
 @GET
 @Produces("text/html")
 public String getHtml() {
   return "<html lang=\"en\"><body><h1>Hello,
          World!!</h1></body></html>";
 }
}
```

Download a JAX-RS implementation



URI Path Templates

- `@Path("/users/{username}")`
  - In this kind of example, a user is prompted to type his or her name, and then a JAX-RS web service configured to respond to requests to this URI path template responds.
  - For example, if the user types the user name "Galileo," the web service responds to the following URL:

    http://example.com/users/Galileo

  - To obtain the value of the user name, the @PathParam annotation may be used on the method parameter of a request method, as shown in the following code example:

```
@Path("/users/{username}")
public class UserResource {
    @GET
    @Produces("text/xml")
    public String getUser(@PathParam("username")  String userName)
    { … }
}
```

Responding to HTTP Methods and Requests

- JAX-RS defines a set of request method designators for the common HTTP methods GET, POST, PUT, DELETE, and HEAD;
  - you can also create your own custom request method designators. Creating custom request method designators is outside the scope of this document.
- The following example shows the use of the PUT method to create or update a storage container:

```
@PUT
public Response putContainer() {
    System.out.println("PUT CONTAINER " + container);
    URI uri = uriInfo.getAbsolutePath();
    Container c = new Container(container, uri.toString());
    Response r;
    if (!MemoryStore.MS.hasContainer(c)) {
        r = Response.created(uri).build(); }
    else {
        r = Response.noContent().build();
    }
    MemoryStore.MS.createContainer(c);
    return r;
}
```

Type Mapping

Java Type	Supported Media Types
`byte[]`	All media types (`*/*`)
`java.lang.String`	All text media types (`text/*`)
`java.io.InputStream`	All media types (`*/*`)
`java.io.Reader`	All media types (`*/*`)
`java.io.File`	All media types (`*/*`)
`javax.activation.DataSource`	All media types (`*/*`)
`javax.xml.transform.Source`	XML media types (`text/xml`, `application/xml`, and `application/*+xml`)
`javax.xml.bind.JAXBElement` and application-supplied JAXB classes	XML media types (`text/xml`, `application/xml`, and `application/*+xml`)
`MultivaluedMap<String, String>`	Form content (`application/x-www-form-urlencoded`)
`StreamingOutput`	All media types (`*/*`), `MessageBodyWriter` only

Type Mapping

- The following example shows how to use MessageBodyReader with the @Consumes and @Provider annotations:

```
@Consumes("application/x-www-form-urlencoded")
@Provider public class FormReader implements
        MessageBodyReader<NameValuePair> {
```

- The following example shows how to use MessageBodyWriter with the @Produces and @Provider annotations:

```
@Produces("text/html")
@Provider public class FormWriter implements
        MessageBodyWriter<Hashtable<String, String>> {
```

Extracting Request Parameters

- You can extract the following types of parameters for use in your resource class:
  - Query, URI path, Form, Cookie, Header, Matrix

- Query parameters

```
@Path("smooth")
@GET
public Response smooth(
    @DefaultValue("2") @QueryParam("step") int step,
    @DefaultValue("true") @QueryParam("min-m") boolean hasMin,
    @DefaultValue("true") @QueryParam("max-m") boolean hasMax,
    @DefaultValue("true") @QueryParam("last-m") boolean hasLast,
    @DefaultValue("blue") @QueryParam("min-color") ColorParam minColor,
    @DefaultValue("green") @QueryParam("max-color") ColorParam maxColor,
    @DefaultValue("red") @QueryParam("last-color") ColorParam lastColor )
{ … }
```

Extracting Request Parameters

- You can extract the following types of parameters for use in your resource class:
  - Query, URI path, Form, Cookie, Header, Matrix

- URI path parameters

```
@Path("/{username}")
public class MyResourceBean {
   ...
   @GET
   public String printUsername(@PathParam("username") String userId)
   { ... }
}
```

Extracting Request Parameters

- You can extract the following types of parameters for use in your resource class:
  - Query, URI path, Form, Cookie, Header, Matrix

- Cookie parameters,
  - indicated by decorating the parameter with javax.ws.rs.CookieParam, extract information from the cookies declared in cookie-related HTTP headers.
- Header parameters,
  - indicated by decorating the parameter with javax.ws.rs.HeaderParam, extract information from the HTTP headers.
- Matrix parameters,
  - indicated by decorating the parameter with javax.ws.rs.MatrixParam, extract information from URL path segments.
- Form parameters,
  - indicated by decorating the parameter with javax.ws.rs.FormParam, extract information from a request representation that is of the MIME media type application/x-www-form-urlencoded and conforms to the encoding specified by HTML forms.

Extracting Request Parameters

- You can extract the following types of parameters for use in your resource class:
  - Query, URI path, Form, Cookie, Header, Matrix

```
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(@FormParam("name") String name) { // Store the message }

@GET
public String get(@Context UriInfo ui) {
   MultivaluedMap<String, String> queryParams = ui.getQueryParameters();
   MultivaluedMap<String, String> pathParams = ui.getPathParameters();
}

@GET
public String get(@Context HttpHeaders hh) {
   MultivaluedMap<String, String> headerParams = hh.getRequestHeaders();
   Map<String, Cookie> pathParams = hh.getCookies();
}
```

Configuring JAX-RS Applications

- A JAX-RS application consists of at least one resource class packaged within a WAR file.
  - The base URI from which an application's resources respond to requests can be set one of two ways:
    - Using the @ApplicationPath annotation in a subclass of javax.ws.rs.core.Application packaged within the WAR
    - Using the servlet-mapping tag within the WAR's web.xml deployment descriptor

```
@ApplicationPath("/webapi")
public class MyApplication extends Application {
    @Override
    public Set<Class<?>> getClasses() {
        final Set<Class<?>> classes = new HashSet<>();
        // register root resource
        classes.add(MyResource.class);
        return classes;
    }
}
```

Configuring JAX-RS Applications

- A JAX-RS application consists of at least one resource class packaged within a WAR file.
  - The base URI from which an application's resources respond to requests can be set one of two ways:
    - Using the @ApplicationPath annotation in a subclass of javax.ws.rs.core.Application packaged within the WAR
    - Using the servlet-mapping tag within the WAR's web.xml deployment descriptor

```
<servlet-mapping>
    <servlet-name>javax.ws.rs.core.Application</servlet-name>
    <url-pattern>/webapi/*</url-pattern>
</servlet-mapping>
```

    - This setting will also override the path set by @ApplicationPath when using an Application subclass.

```
<servlet-mapping>
    <servlet-name>com.example.rest.MyApplication</servlet-name>
    <url-pattern>/services/*</url-pattern>
</servlet-mapping>
```

Accessing REST Resources

- The following steps are needed to access a REST resource using the Client API.
  - Obtain an instance of the javax.ws.rs.client.Client interface.
  - Configure the Client instance with a target.
  - Create a request based on the target.
  - Invoke the request.

```
Client client = ClientBuilder.newClient();
String name = client.target("http://example.com/webapi/hello")
                    .request(MediaType.TEXT_PLAIN)
                    .get(String.class);
```

Accessing REST Resources

- Setting Path Parameters in Targets

```
WebTarget myResource = client.target("http://example.com/webapi/read")
                                .path("{userName}");
Response response = myResource.queryParam("userName", "janedoe")
                              .request(...)
                              .get();
```

- Supported methods are:
  - get()
  - post()
  - delete()
  - put()
  - head()
  - options()

Trade-off of WS

- Advantages:
  - Across platforms
    - XML-based, independent of vendors
  - Self-described
    - WSDL: operations, parameters, types and return values
  - Modulization
    - Encapsulate components
  - Across Firewall
    - HTTP
- Disadvantages:
  - Lower productivity
    - Not suitable for stand-alone applications
  - Lower performance
    - Parse and assembly
  - Security
    - Depend on other mechanism, such as HTTP+SSL

When

- When we should use WS:
  - Support communication across firewall
  - Support application integration
  - Support B2B integration
  - Encourage reusing software
- When we should NOT use WS:
  - Stand-alone applications
    - Such as MS Office
  - Homogeneous applications in LAN
    - Such as communication between COM+s or EJBs

Project

- 3rd Iteration Requirement
  - Applying MemCached or Redis in your application.
  - Refactoring your application to support internationalization.
  - Writing a design doc to describe the clustering solution you will adopt to scale up your Book Store.
  - Developing an Aspect to implement logging.
  - Respectively developing a SOAP and a REST web service for a query.

- Deadline 5.25 0:00

References

- Core Java (volume II) 9th edition
  - http://horstmann.com/corejava.html
- The Java EE 7 Tutorial
  - http://docs.oracle.com/javaee/7/tutorial/doc/javaeetutorial7.pdf
- 如何获取（GET）一杯咖啡——星巴克REST案例分析
  - http://tech.ddvip.com/2008-12/122836047297260.html

Thank You!