RESTful Web Services

Week 9 lecture

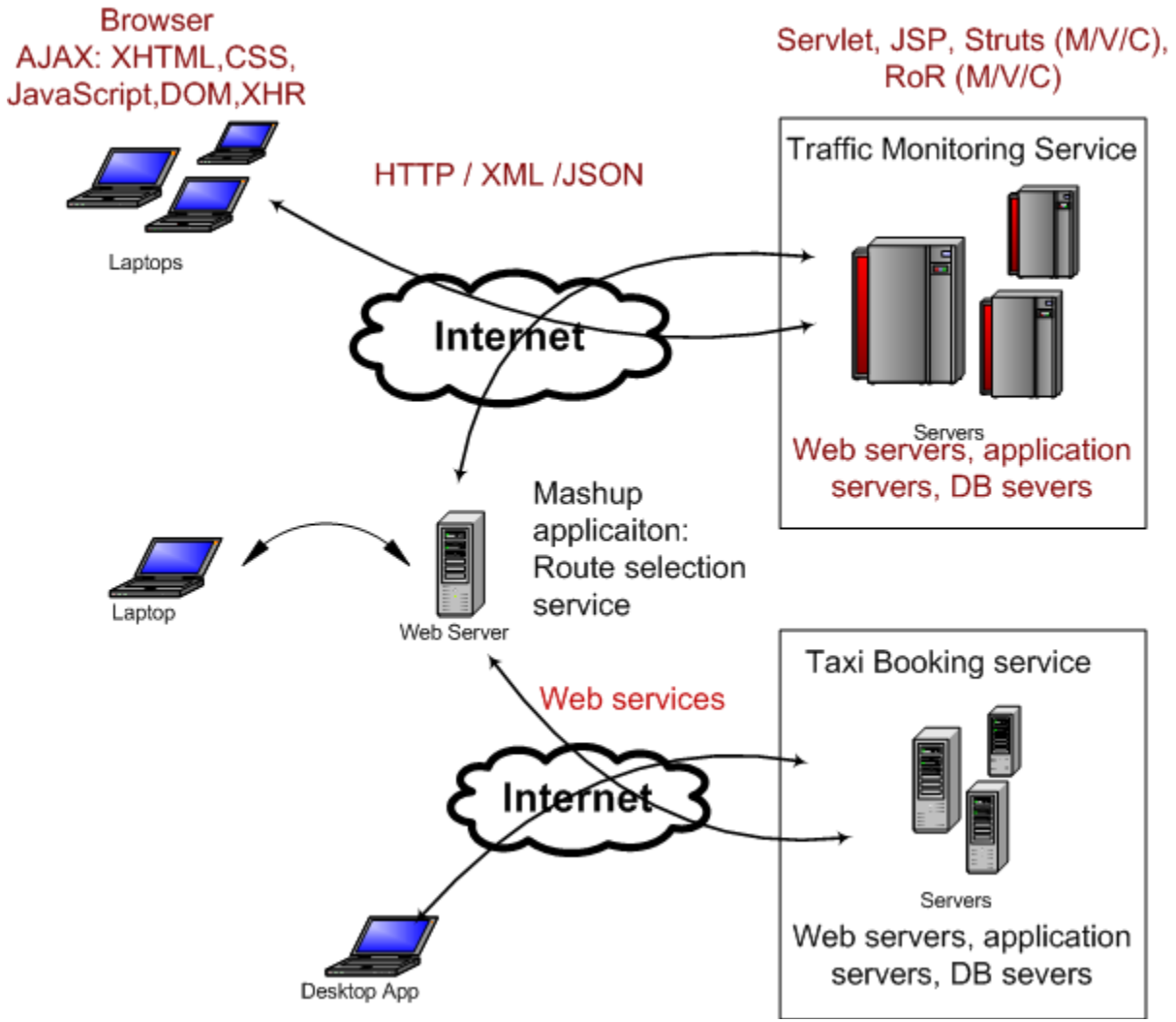
Agenda

- Examples of published public web services
  - The so called SOAP vs. REST
- Java API for RESTful Web Services (JAX-RS)
- Assignment 2 intro

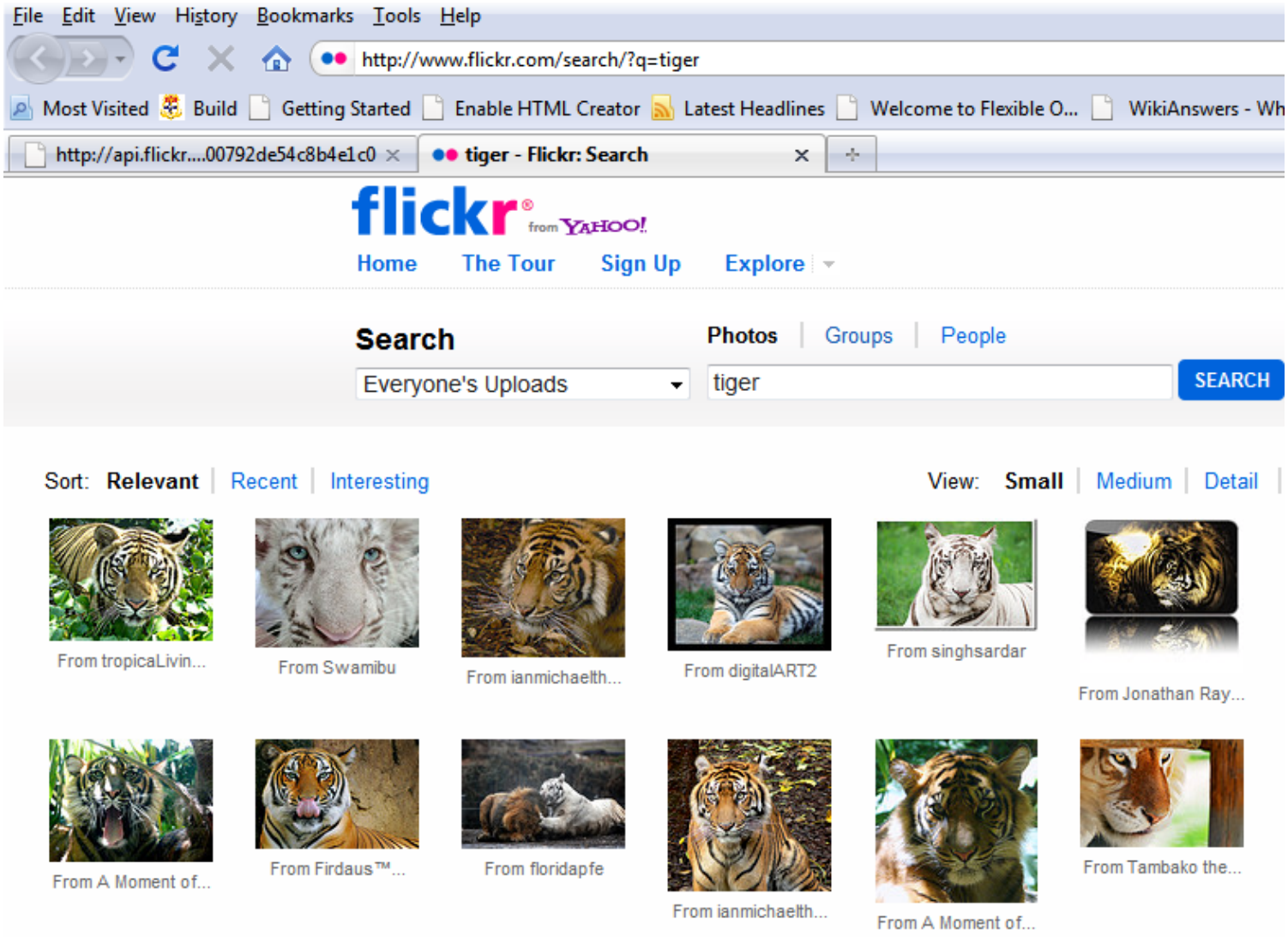
What are web services

- Web services is a distributed architectural paradigm for applications
- It provides a simple and open way of integrating functions or data from various systems
- It can be used within an organization and across the public Internet
- When it was first proposed, it consists of several basic standards
  - SOAP: A messaging protocol for transferring information
  - WSDL: A model and an XML format for describing Web services
  - UDDI: A registry and protocol for publishing and discovering web services (**not really used!!)**
  - WSDL and UDDI are in tension with the idea of using URI to address web resources
  - Original design of Web Services is very application centric **in contrast** to the resource centric Web and REST style.
- The term web services has much broader meaning now
  - At least two implementations: SOAP based vs. RESTful

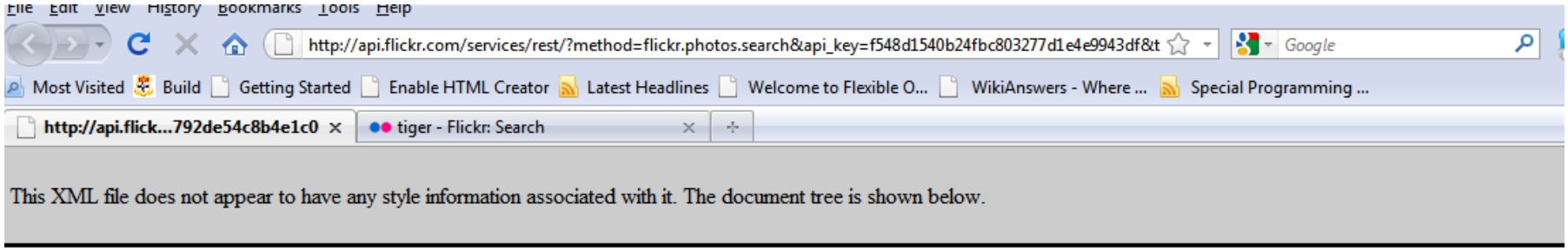
What can we use web services for



Regular browser generated query



Web services query




```
−<rsp stat="ok">
  −<photos page="1" pages="11162" perpage="100" total="1116161">
      <photo id="4600568402" owner="45973339@N06" secret="d1c05088d0" server="4005" farm="5" title="Kids and Tigers" ispublic="1" isfriend="0" isfamily="0"/>
      <photo id="4367861856" owner="76459353@N00" secret="95830e136d" server="4023" farm="5" title="No Racism Please" ispublic="1" isfriend="0" isfamily="0"/>
      <photo id="4600559828" owner="77541634@N00" secret="3bf473b658" server="1285" farm="2" title="Tiger" ispublic="1" isfriend="0" isfamily="0"/>
      <photo id="4599941883" owner="77541634@N00" secret="ac46c4efa0" server="1146" farm="2" title="Tiger" ispublic="1" isfriend="0" isfamily="0"/>
      <photo id="4599909787" owner="26837544@N03" secret="14f7cac95e" server="3550" farm="4" title="Peek-A-Boo" ispublic="1" isfriend="0" isfamily="0"/>
      <photo id="4600522370" owner="15339304@N06" secret="5845f04cbc" server="1048" farm="2" title="IMG_7628" ispublic="1" isfriend="0" isfamily="0"/>
      <photo id="4598951328" owner="40933877@N00" secret="becca69f71" server="1091" farm="2" title="~ Eye Of the Tiger ~" ispublic="1" isfriend="0" isfamily="0"/>
      <photo id="4599903547" owner="15339304@N06" secret="0d1c1640ce" server="4014" farm="5" title="IMG_7626" ispublic="1" isfriend="0" isfamily="0"/>
      <photo id="4599896155" owner="38160534@N02" secret="fbf9ae15cc" server="1430" farm="2" title="" ispublic="1" isfriend="0" isfamily="0"/>
      <photo id="4599903573" owner="15339304@N06" secret="d7a46163db" server="1361" farm="2" title="IMG_7627" ispublic="1" isfriend="0" isfamily="0"/>
      <photo id="4600469458" owner="28738131@N08" secret="95372459e7" server="1085" farm="2" title="Gaur | Bos gaurus" ispublic="1" isfriend="0" isfamily="0"/>
      <photo id="4599862041" owner="51521377@N00" secret="058e43af1b" server="1080" farm="2" title="Bee Eaters of the 3rd kind!" ispublic="1" isfriend="0"
      isfamily="0"/>
```


http://api.flickr.com/services/rest/?method=flickr.photos.search&
api_key=f548d1540b24fbc803277d1e4e9943df&text=tiger&api_sig=546727ab6596ffb00792de54c8b4e1c0

Example Web Service APIs

- Yahoo API:
  - http://developer.yahoo.com/everything.html
- Flickr API
  - http://www.flickr.com/services/api/
- Amazon product advertising API
  - https://affiliate-program.amazon.com/gp/advertising/api/detail/main.html#details
- New York Times API
  - http://developer.nytimes.com/docs
- Youtube API
  - https://developers.google.com/youtube/getting_started#data_api
- Ebay API
  - http://developer.ebay.com/common/api/
  - http://developer.ebay.com/DevZone/finding/CallRef/findItemsByKeywords.html

SOAP vs. REST: Superficial difference

- What many programmers see, what most public web services implement, and what largely determines the fate of both
- As a client/consumer of web services
  - To consume a REST style web services
    - The request is just a normal HTTP request where you can include the parameters in query strings or request body
    - The response is just a normal XML (or JSON) document
    - Straightforward implementation
  - To consume a SOAP style web services
    - The request should contain a body of XML document called a SOAP request message, it has to follow certain XML schema
    - The response is also a SOAP response message
    - We normally use certain package to handle SOAP message
      - Javax.xml.soap, Apache Axis, …

Example SOAP client

- General information **get** from the Flickr website
  - The SOAP Server Endpoint URL is http://api.flickr.com/services/soap/
  - The SOAP Request looks like

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
         xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
         xmlns:xsd="http://www.w3.org/1999/XMLSchema" >
    <s:Body>
       <x:FlickrRequest xmlns:x="urn:flickr">
            <method>flickr.test.echo</method>
           <name>value</name>
      </x:FlickrRequest>
   </s:Body>
</s:Envelope>
```

From http://www.flickr.com/services/api/request.soap.html

The Java SOAP client would look like

```
public void callPanda(String pandaName){
    try{
            MessageFactory mf = MessageFactory.newInstance();
            SOAPFactory sf = SOAPFactory.newInstance();
            SOAPMessage msg = mf.createMessage();
            SOAPPart sp = msg.getSOAPPart();
            SOAPEnvelope env = sp.getEnvelope();
            env.addNamespaceDeclaration("xsi","http://www.w3.org/1999/XMLSchema-instance");
            env.addNamespaceDeclaration("xsd","http://www.w3.org/1999/XMLSchema");
            SOAPBody bd = env.getBody();
            // create the main element of the body
            SOAPElement be = sf.createElement(env.createName("FlickrRequest","x","urn:flickr"));
            // create the method element
            SOAPElement method = sf.createElement("method");
            method.setValue(Constants.METHOD); // it is flickr.panda.getPhotos here
            // create the parameter "panda_name"
            SOAPElement pandaNamePara=sf.createElement("panda_name");
            pandaNamePara.setValue(pandaName);
            // create the api_key element
            SOAPElement apiKey = sf.createElement("api_key");
            apiKey.setValue(Constants.API_KEY);
```

Prepare the soap request message content

```
            // link them together
            be.addChildElement(method);
            be.addChildElement(pandaNamePara);
            be.addChildElement(apiKey);
            bd.addChildElement(be);
```

Link content in required format

```
            //create the connection and call the service
            SOAPConnection conn = SOAPConnectionFactory.newInstance().createConnection();
            SOAPMessage response= conn.call(msg, Constants.SOAP_ENDPOINT);
```

Call the SOAP web services

```
    }catch (Exception e){
            System.out.println(e.getMessage());
    }
}
```

Example REST client

- General information
  - The REST Endpoint URL is http://api.flickr.com/services/rest/
  - The url sent for panda search would be **http://api.flickr.com/services/rest/?method=flickr.panda.getPhotos&panda_name=ling+ling&api_key=somekey**
  - Example response

```
<photos interval="60000" lastupdate="1235765058272" total="120" panda="ling ling">
     <photo title="Shorebirds at Pillar Point" id="3313428913"
            secret="2cd3cb44cb" server="3609" farm="4" owner="72442527@N00" ownername="Pat Ulrich"/>
    <photo title="Battle of the sky" id="3313713993"
            secret="3f7f51500f" server="3382" farm="4" owner="10459691@N05" ownername="Sven Ericsson"/>
</photos>
```

The Java REST client would look like

This is the REST request!

```
public void callPanda(String pandaName){
    try{
            String callUrlStr = Constants.REST_ENDPOINT+"?method="+Constants.METHOD+
                        "&panda_name="+URLEncoder.encode(pandaName,Constants.ENC)+
                        "&per_page="+Constants.DEFAULT_NUMBER+
                        "&api_key="+Constants.API_KEY;

            URL callUrl = new URL(callUrlStr);
            HttpURLConnection urlConnection = (HttpURLConnection)callUrl.openConnection();
            InputStream urlStream = urlConnection.getInputStream();

            DocumentBuilder db = DocumentBuilderFactory.newInstance().newDocumentBuilder();
            Document response = db.parse(urlStream);
                                                    Call the REST service
            //print out all titles
            System.out.println("The titles returned are: ");
            NodeList nl = response.getElementsByTagName("photo");
            for (int i = 0; i < nl.getLength(); i ++){
                        System.out.println(nl.item(i).
                                    getAttributes().getNamedItem("title").getTextContent());
            }
            urlConnection.disconnect();
    }catch (Exception e){
            System.out.println(e.getMessage());
    }
}
```

The truth is…

- Many published REST format web services are not truly RESTful
  - Use only the GET and POST method
  - Resources are not properly expressed in URI
  - You can see some SOAP flavour there
  - But most of them follow the requirement that HTTP messages should be as *self-descriptive* as possible
- Many published SOAP Web services format only give you a small and biased view of what SOAP Web Service really is
  - SOAP is a protocol, it runs on top of many other protocols, not just HTTP
  - SOAP can be web friendly
  - SOAP messages can be exchanged using HTTP GET method

Why SOAP feels so heavy?

- Flickr's SOAP API only uses the SOAP message format
  - It does not have an WSDL to define the input/output message and respective types of elements inside the message
  - It does not have any control element in the SOAP header
- Amazon, Google and Ebay's SOAP APIs have more to show
  - Proper WSDL
  - Proper way to generate proxy and skeleton on both sides
- Most published public web services provide relative simple "services" on relatively simple types of web resources
  - No need to add additional control on SOAP header
  - No need to provide WSDL
    - Most xml request/response message are self-descriptive enough
    - Programmers can "guess" the semantics and syntax of the message easily
    - It is quite easy to handle custom format XML/JSON message
  - SOAP does not add any obvious benefits for its obvious heaviness compared with REST

What is REST

- **Representational State Transfer**
- REST-style architectures consist of clients and servers. Clients initiate requests to servers; servers process requests and return appropriate responses. Requests and responses are built around the transfer of representations of resources. A resource can be essentially any coherent and meaningful concept that may be addressed. A representation of a resource is typically a document that captures the current or intended state of a resource.

Based on Roy Fielding's doctoral dissertation, rephrased by wikipedia
http://en.wikipedia.org/wiki/Representational_State_Transfer

SOAP and RESTful is not mutually exclusive

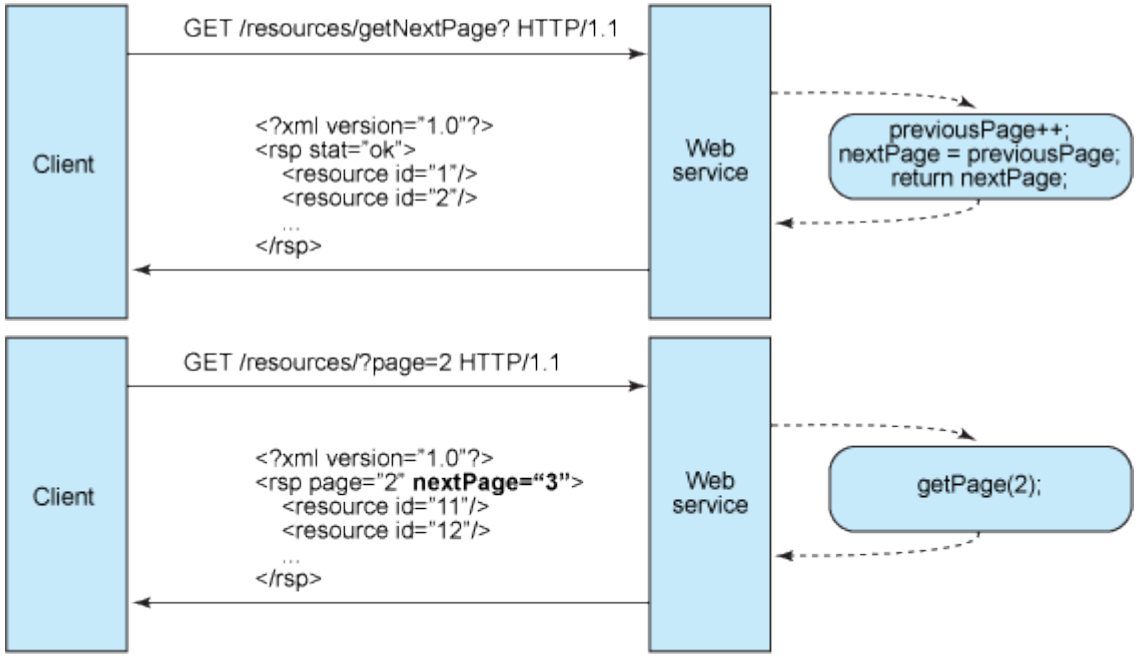
- SOAP is a protocol while RESTful is more or less a style
- SOAP comes from traditional distributed system area
  - Both Amazon and Google have strong tradition in distributed computing
  - The whole set SOAP web services technologies has lots of features to enable services description and integration
- RESTful style also comes from distribution system area
  - Has Web flavour in it
  - We only discuss the RESTful style adapted in a few web application framework

Web Architecture Compatibility

- A URI as a resource identifier is one of the central concepts of WWW
  - A predominant use of the World Wide Web is pure information retrieval, where the representation of an available resource, identified by a URI, is fetched using a HTTP GET request without affecting the resource in any way.
- The simplicity and scalability of the Web is largely due to the fact that there are a few "generic" methods (GET, POST, PUT, DELETE) which can be used to interact with any resource made available on the Web via a URI.
- RESTful style is compatible with general web architecture (Web friendly)
- It is recommended that resources that may be accessed by SOAP should, where practical, place any such resource-identifying information as a part of the URI identifying the target of the request

Basic REST design principles

- Use HTTP methods explicitly
- Be stateless
  - Address the resources explicitly in the request message



- Expose directory structure-like URIs
  - http://www.myservice.org/discussion/topics/{topic}
- Transfer XML, JSON, or both

http://www.ibm.com/developerworks/webservices/library/ws-restful/

Creating Rest Service using J2EE 6

- J2EE 6 includes JAX-RS API for creating RESTful web services
- Jersey is the reference implementation of the JAX-RS
- Several application servers have Jersey included
- To deploy Jersey based RESTful web services on Tomcat, you need to include Jersey library
- JAX-RS relies on java annotations to implement RESTful web services
  - It has annotations to specify location of **resources**, url mappings to various resource manipulation methods, xml/json format conversion for resources

Important Concepts

- Resources
  - Root Resources
    - Some resources that you obtain directly
  - Sub Resources
    - Related with a root resource, is obtained through the root resource
  - Example
    - The collection of Products can be viewed as a root resource while individual Product can be viewed as a sub resource
    - Movie can be viewed as a root resource while Actors belonging to the movie is a sub resource
  - *Root resource classes* are POJOs (Plain Old Java Objects) that are annotated with @Path have at least one method annotated with @Path or a *request method designator* annotation such as @GET, @PUT, @POST, or @DELETE.
- Resource methods
  - are methods of a resource class annotated with a request method designator.

A Root Resource Example

```
package rest;
/**
 * Representing the ProductResource in Rest web Services
 */
@Path("/products")
public class ProductResource {
    private ProductDao pdao = DaoFactory.getInstance().getProductDao();
    @Context UriInfo uriInfo; //like an instance variable definition

    @GET
    @Produces(MediaType.TEXT_XML)
    //handles methods like http://localhost:8080/shoppingCart/rest/products
    public List<Product> getAllProducts() {
          return pdao.getAllProducts();
    }

    @GET
    @Path("{id}")
    @Produces(MediaType.TEXT_XML)
    //handles methods like http://localhost:8080/shoppingCart/rest/products/1
    public Product getProductById(@PathParam(value="id") int productId) {
          return pdao.getProductById(productId);
    }
    …
```

Important annotations

- @Path annotation
  - The @Path annotation's value is a ***relative*** URI path.
    - The base URL is based on your application name, the servlet and the URL pattern from the web.xml configuration file
    - http://localhost:8080/shoppingCart/rest**/products**
  - Variables can be embedded in @Path value; variables are denoted by curly braces;
  - Variable values are obtained through the @PathParam annotation, which may be used on method parameter of a request method
- *request method designator*
  - @GET, @PUT, @POST , @DELETE, and @HEAD are *request method designator* annotations defined by JAX-RS and which correspond to the similarly named HTTP methods.

Important annotations – cont'd

- **@Produces**
  - is used to specify the MIME media types of representations a resource can produce and send back to the client.
- **@Consumes**
  - is used to specify the MIME media types of representations a resource can consume that were sent by the client.
- **@Context**
  - Is used to obtain information like HTTP header, request, uri and so on
- **Representations and Java types**
  - JAX-RS supports the automatic creation of XML and JSON via JAXB
  - @XmlRootElement can automatically convert the annotated java class into xml format

The model class

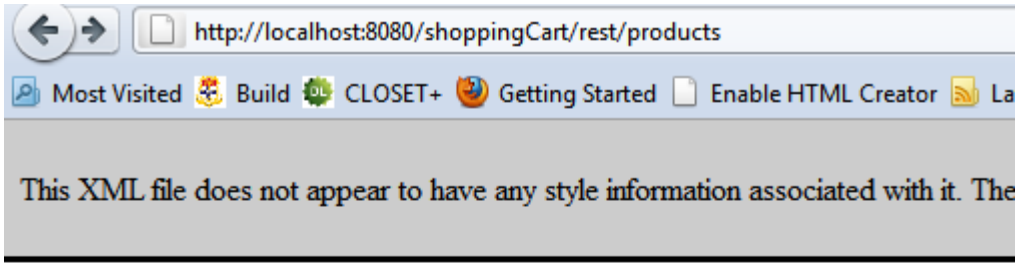
```
package model;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Product {
    private String title,description,imageUrl;
    private double price;
    private int productId = -1; //default id will be modified by the storage
…
}
```

Deploying a RESTful web services

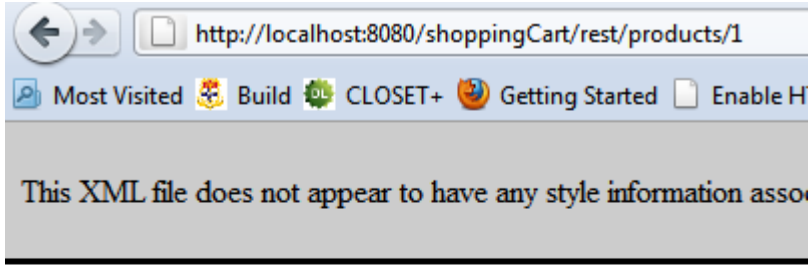
- Many different ways to deploy the services
- Relying on a Servlet provided by implementation framework

```
<servlet>
        <servlet-name>ProductRest</servlet-name>

    <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
    <init-param>
      <param-name>com.sun.jersey.config.property.packages</param-name>
      <param-value>rest</param-value>
    </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>ProductRest</servlet-name>
        <url-pattern>/rest/*</url-pattern>
    </servlet-mapping>
```

JAX-RS RESTful request through browser




```
−<products>
  −<product>
      <description>textbook</description>
      <imageUrl>../imgs/webapplication.jpg</imageUrl>
      <price>79.95</price>
      <productId>1</productId>
      <title>Web Application Architecture</title>
    </product>
  −<product>
      <description>textbook</description>
      <imageUrl>../imgs/wwwprogramming.jpg</imageUrl>
      <price>109.95</price>
      <productId>2</productId>
      <title>Internet How to Program</title>
    </product>
</products>
```




```
−<product>
    <description>textbook</description>
    <imageUrl>../imgs/webapplication.jpg</imageUrl>
    <price>79.95</price>
    <productId>1</productId>
    <title>Web Application Architecture</title>
  </product>
```

RESTful POST request

```
@POST
@Consumes(MediaType.APPLICATION_XML)
public Response addProduct(JAXBElement<Product> productXML) {
    Response res = null;
    Product product = productXML.getValue();
    int productId = product.getProductId();
    if (productId != -1){ //an old resource
          pdao.updateProduct(product);
          res =Response.noContent().build();
    }else{
          pdao.addProduct(product);
          res = Response.created(uriInfo.getAbsolutePath()).build();
    }
    return res;
}
```

Creating simple java client

```
// Omit package and import statements
public class ProductRestClient {
    public static void main(String[] args) {
        ClientConfig config = new DefaultClientConfig();
        Client client = Client.create(config);
        WebResource service = client.resource(getBaseURI());
        // Create one product
        Product product = new Product("Shutter Island","Novel","../imgs/webapplication.jpg",14.95);
        ClientResponse response =
        service.path("rest").path("products").accept(MediaType.APPLICATION_XML).post(ClientResponse.class,
        product);
        // Return code should be 201 == created resource
        System.out.println(response.getStatus());
        // Update product with id 1
        product = new Product(1, "Web Application Architecture","textbook","../imgs/webapplication.jpg",50.95);
        response =
    service.path("rest").path("products").accept(MediaType.APPLICATION_XML).post(ClientResponse.class, product);
        // Return code should be 204 == no content
        System.out.println(response.getStatus());
        // Get product with id 1
        System.out.println(service.path("rest").path("products/1").accept(
                    MediaType.TEXT_XML).get(String.class));
    }
    private static URI getBaseURI() {
        return UriBuilder.fromUri("http://localhost:8080/shoppingCart").build();
    }
}
```

After running java service client

Console output:

```
201
204
<?xml version="1.0" encoding="UTF-8"
standalone="yes"?><product><description>textbook
</description><imageUrl>../imgs/webapplication.j
pg</imageUrl><price>50.95</price><productId>1</p
roductId><title>Web Application
Architecture</title></product>
```

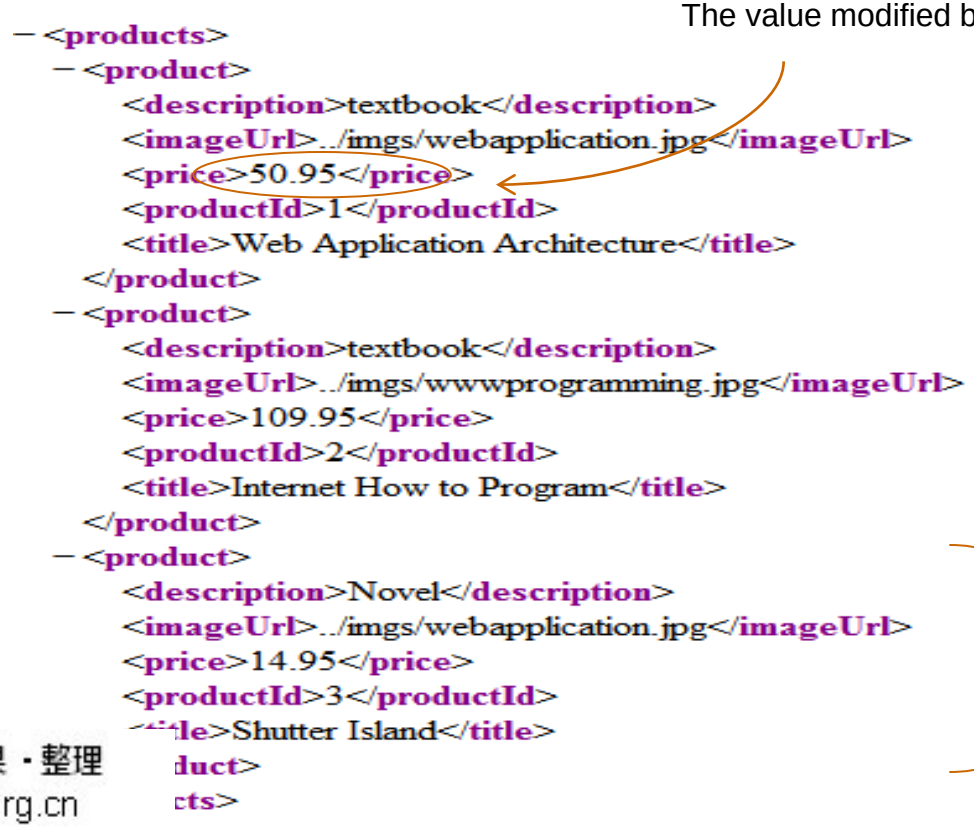
The response from the first POST request

The response from the second POST request

The response from the GET request

Browser client output:

The value modified by java client



New product added by java client