REST

- Architectural style for web APIs originally suggested by Roy Fielding (http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm)
- REpresentational
  - Use any form of representation for the resources (XML, JSON, HTML, etc) - whatever best suits to the purpose, even multiple
- State
  - Primary concern is the state of the resource, not operations performed on it
- Transfer
  - Resource data is transferred from an application to another

REST - identify resources with URLs

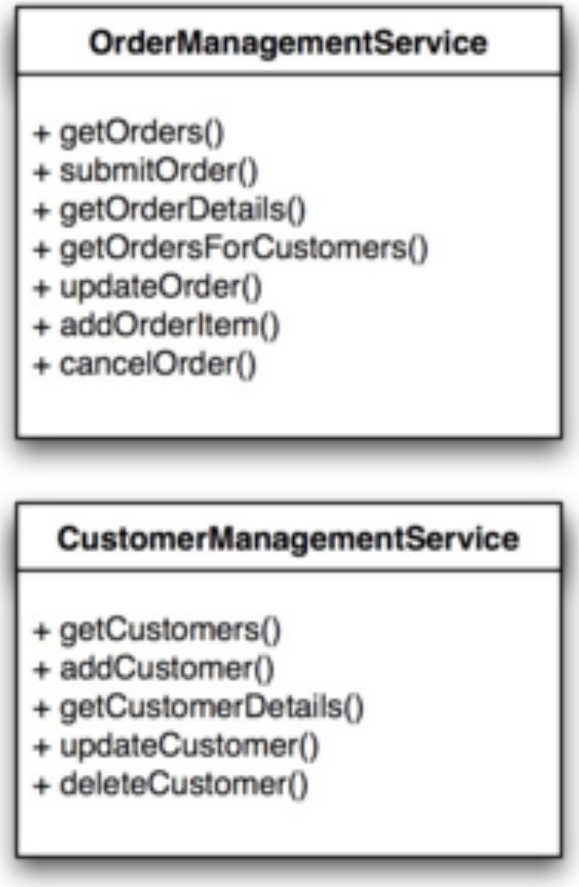
- RESTless URL:
  - `http://localhost:8080/Spitter/displaySpittle.htm?id=87`
- RESTful URL
  - `http://localhost:8080/Spitter/spittles/87`
  - Hierarchical URL for addressing the resource
  - Omit /87 and you have an URL for all spittles, for example
- But… what is supposed to be done with the resource?

Restful HTTP - REST and HTTP methods

- GET
  - retrieve resource from the server without changing resource state, is idempotent
- POST
  - posts (adds) data to server, changes state and is not idempotent
- PUT
  - puts (updates) resource data in the server, changes state and is not idempotent
- DELETE
  - delete the resource, changes the state and is idempotent
- OPTIONS
  - request options, doesn’t change state and is idempotent

Example case

- From http://www.infoq.com/articles/rest-introduction



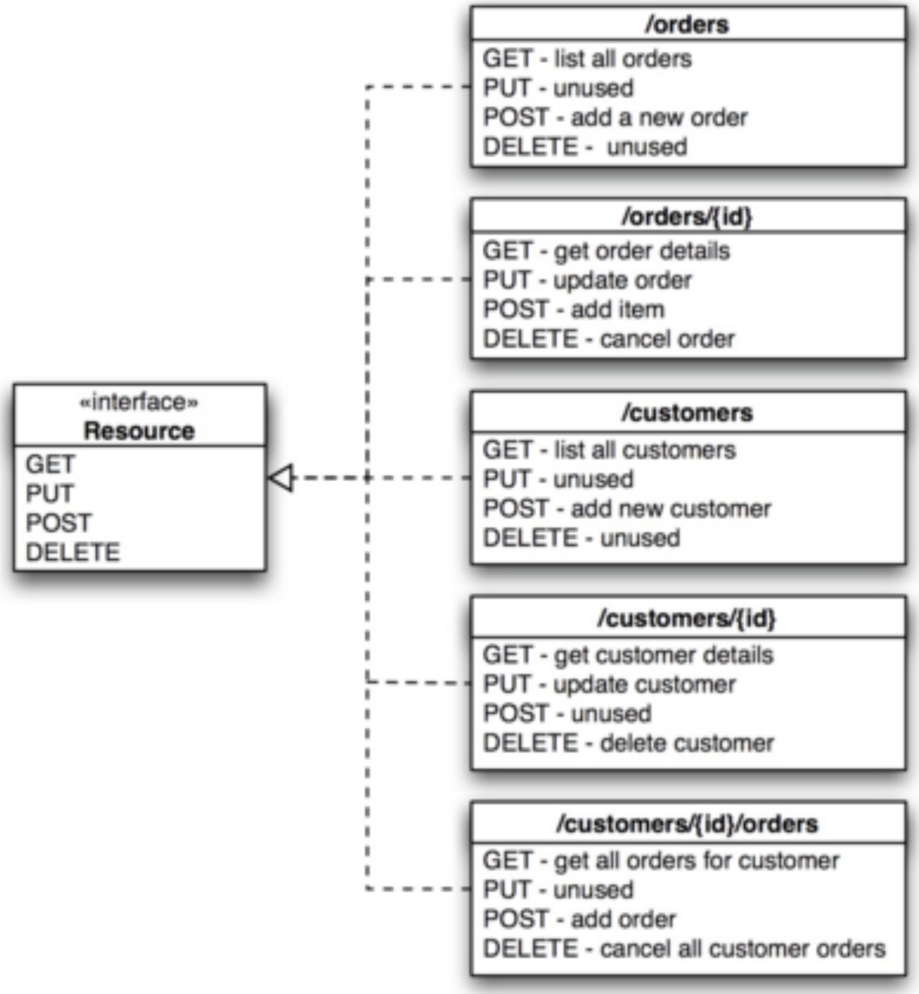
OrderManagementService

+ getOrders()
+ submitOrder()
+ getOrderDetails()
+ getOrdersForCustomers()
+ updateOrder()
+ addOrderItem()
+ cancelOrder()

CustomerManagementService

+ getCustomers()
+ addCustomer()
+ getCustomerDetails()
+ updateCustomer()
+ deleteCustomer()

Example API



«interface»
Resource

GET
PUT
POST
DELETE

/orders

GET - list all orders
PUT - unused
POST - add a new order
DELETE - unused

/orders/{id}

GET - get order details
PUT - update order
POST - add item
DELETE - cancel order

/customers

GET - list all customers
PUT - unused
POST - add new customer
DELETE - unused

/customers/{id}

GET - get customer details
PUT - update customer
POST - unused
DELETE - delete customer

/customers/{id}/orders

GET - get all orders for customer
PUT - unused
POST - add order
DELETE - cancel all customer orders

Implement in server

- Process all requests with own code
  - Parse URLs, process all relevant HTTP methods per URL pattern
  - Not very straightforward in case of a large system
- Use a framework
  - Multiple available, we'll use Jersey
  - https://jersey.java.net/documentation/latest/user-guide.html - esp. chapter 3

Jersey

- Jersey framework is included in Netbeans + Glassfish bundle
- Create Java web application (like in tutorials)
  - Create "normal" Java classes for your application logic
  - Create "resource" classes to represent resources your API exposes.
  - Implement in those classes needed operations (GET, PUT, POST, etc)
  - Note: resource class is by default instantiated for each operation. To refer to model objects you might want to use singleton pattern (in model)

Example snippets

```java
@Path("/Teams")
public class TeamsResource {
    private final SportsWorld world;

    public TeamsResource() {
        this.world = SportsWorld.getInstance();
    }

    @GET
    @Produces("text/plain")
    public String getTeamsPlain() {
        String result = "";
        for(Team t: world.getTeams()) {
            result += "<" + t.getId() + "> ";
        }
        return result;
    }

    @Path("/{teamid}")
    @GET
    @Produces("text/plain")
    public String getTeamPlain(@PathParam("teamid") int teamid) {
        return world.getTeamById(teamid).toString();
    }

}
```

Expose URI "/Teams"

Using singleton model object since TeamsResource will be instantiated for each request.

HTTP GET on /Teams is served here. Produces plain text.

HTTP GET on /Teams/<number> is served here. Produces plain text. Note path parameter is used in method parameter.

Produce XML - model side

```
@XmlRootElement
public class Team {
    private final int id;
    private final String name;
    private final HashMap<Integer, Player> players;

    public Team() {
      …
    }

    public Team(int id, String name) {
        this.id = id;
        this.name = name;
        this.players = new HashMap<>();
    }

    @XmlElement
    public int getId() {
        return this.id;
    }

    @XmlElement
    public String getName() {
        return this.name;
    }
…
```

Root of XML, will name XML element “team” by default

Empty constructor needed!

Annotate getters with @XmlElement

```
<?xml version="1.0" encoding="UTF-8"?>
   <team>
       <id>3</id>
       <name>Tottenham</name>
   </team>
```

Produce XML - resource side

```
@Path("/Teams")
public class TeamsResource {
    private final SportsWorld world;

    public TeamsResource() {
        this.world = SportsWorld.getInstance();
    }

    @GET
    @Produces("text/plain")
    public String getTeamsPlain() {
        String result = "";
        for(Team t: world.getTeams()) {
            result += "<" + t.getId() + "> ";
        }
        return result;
    }

    @Path("/{teamid}")
    @GET
    @Produces(MediaType.APPLICATION_XML)
    public Team getTeamXML(@PathParam("teamid") int teamid) {
        return world.getTeamById(teamid);
    }
}
```

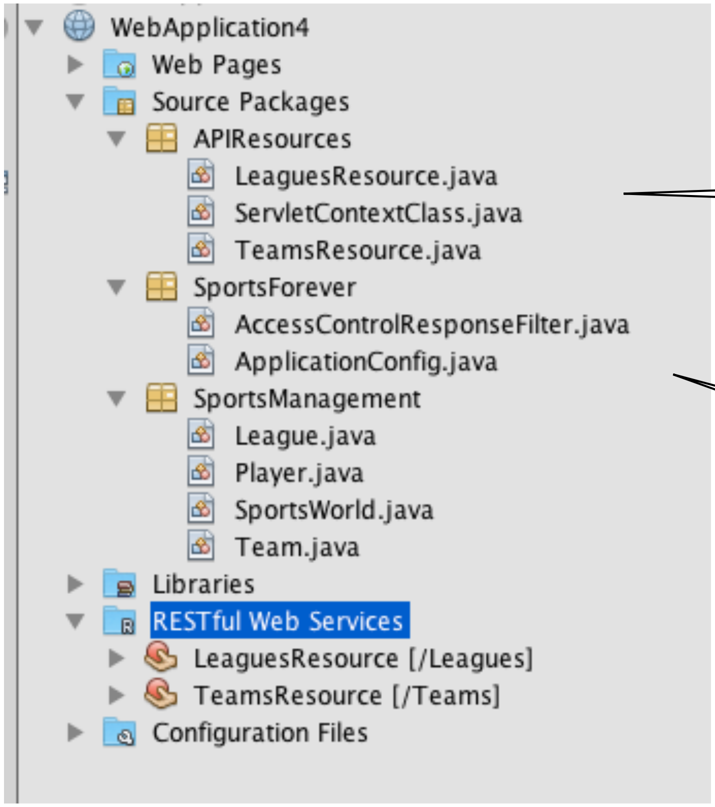
Will convert the Team object returned into XML according to annotations in Team class

Consume XML - resource side

```
@Path("/Testclass")
public class TestClassResource {

    @GET
    @Produces(MediaType.APPLICATION_XML)
    public TestClass getTestClassInstance() {
        return SportsWorld.getInstance().getTest();
    }

    @POST
    @Consumes(MediaType.APPLICATION_XML)
    @Produces(MediaType.APPLICATION_XML)
    public TestClass setTestClassInstance(TestClass tc) {
        SportsWorld.getInstance().setTest(tc);
        return SportsWorld.getInstance().getTest();
    }

}
```

Note that TestClass still needs to have an empty constructor. TestClass instance needs to have setters (and getters) to set the state

Practicalities



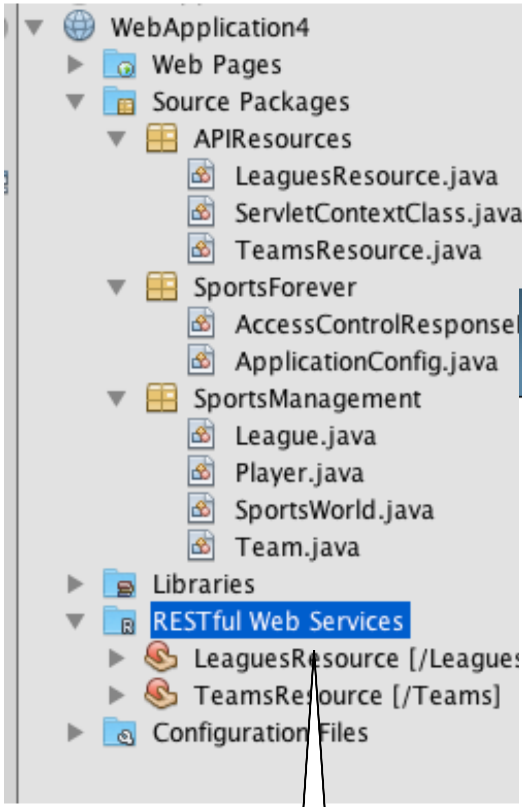
- WebApplication4
  - Web Pages
  - Source Packages
    - APIResources
      - LeaguesResource.java
      - ServletContextClass.java
      - TeamsResource.java
    - SportsForever
      - AccessControlResponseFilter.java
      - ApplicationConfig.java
    - SportsManagement
      - League.java
      - Player.java
      - SportsWorld.java
      - Team.java
  - Libraries
  - RESTful Web Services
    - LeaguesResource [/Leagues]
    - TeamsResource [/Teams]
  - Configuration Files

Using packages will help in not getting confused. Split into API classes and other (esp. model) is a good idea

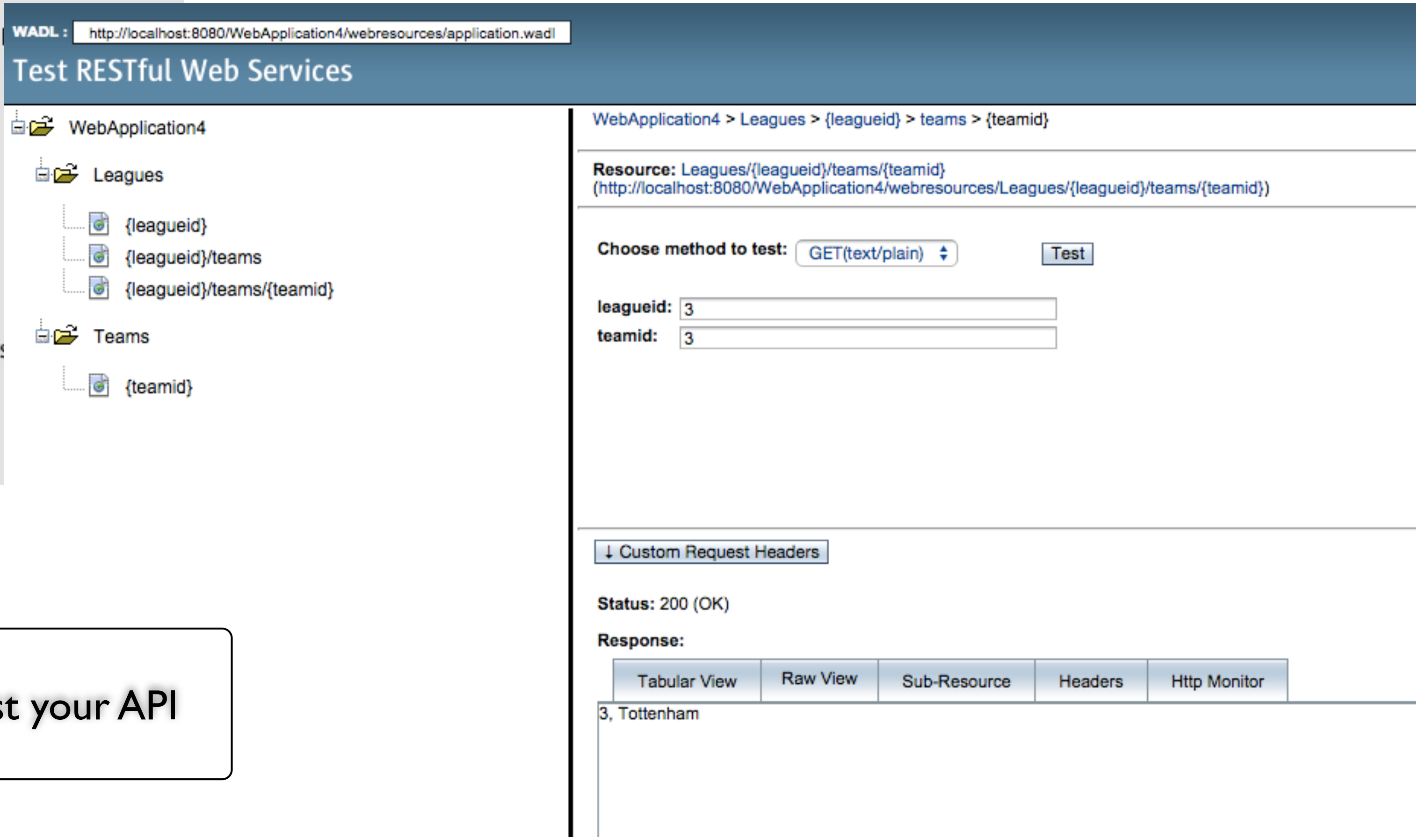
AccessControlResponseFilter to enable cross-site scripting

```
@Provider
@Priority(Priorities.HEADER_DECORATOR)
public class AccessControlResponseFilter implements ContainerResponseFilter {

    @Override
    public void filter(ContainerRequestContext requestContext, ContainerResponseContext responseContext) throws IOException {
        final MultivaluedMap<String,Object> headers = responseContext.getHeaders();

        headers.add("Access-Control-Allow-Origin", "*");
        headers.add("Access-Control-Allow-Headers", "Authorization, Origin, X-Requested-With, Content-Type");
        headers.add("Access-Control-Expose-Headers", "Location, Content-Disposition");
        headers.add("Access-Control-Allow-Methods", "POST, PUT, GET, DELETE, HEAD, OPTIONS");
    }
}
```

Practicalities



- WebApplication4
  - Web Pages
  - Source Packages
    - APIResources
      - LeaguesResource.java
      - ServletContextClass.java
      - TeamsResource.java
    - SportsForever
      - AccessControlResponse
      - ApplicationConfig.java
    - SportsManagement
      - League.java
      - Player.java
      - SportsWorld.java
      - Team.java
  - Libraries
  - RESTful Web Services
    - LeaguesResource [/League
    - TeamsResource [/Teams]
  - Configuration Files



WADL : http://localhost:8080/WebApplication4/webresources/application.wadl

Test RESTful Web Services

- WebApplication4
  - Leagues
    - {leagueid}
    - {leagueid}/teams
    - {leagueid}/teams/{teamid}
  - Teams
    - {teamid}

WebApplication4 > Leagues > {leagueid} > teams > {teamid}

Resource: Leagues/{leagueid}/teams/{teamid}
(http://localhost:8080/WebApplication4/webresources/Leagues/{leagueid}/teams/{teamid})

Choose method to test: GET(text/plain) Test

leagueid: 3
teamid: 3

↓ Custom Request Headers

Status: 200 (OK)

Response:

```
3, Tottenham
```

Right-click here to test your API