RESTful (Web) Applications In Practice

Nupul Kukreja

CS 578

Agenda

- REST – What/Why?
- REST in Practice – How?
  - Real life example ☺
- Architectural styles commonly encountered when creating RESTful (web) systems
  - Event Based
  - MVC (Client/Server)
- Data Abstraction Layers (patterns)
  - Data Mapper
  - Active Record

REST (**RE**presentational **S**tate **T**ransfer)

- The architectural style of the web

  So what %&*@#$ does that even mean??
- REST is a set of design criteria and not the physical structure (architecture) of the system
- REST is not tied to the ‘Web’ i.e. doesn’t depend on the mechanics of HTTP
- ‘Web’ applications are the most prevalent – hence RESTful architectures run off of it

Understanding REST – Uniform Interface

- HTTP Provides 4 basic methods for CRUD (create, read, update, delete) operations:
  - **GET**: Retrieve representation of resource
  - **PUT**: Update/modify existing resource (or create a new resource)
  - **POST**: Create a new resource
  - **DELETE**: Delete an existing resource
- Another 2 less commonly used methods:
  - **HEAD**: Fetch meta-data of representation only (i.e. a metadata representation)
  - **OPTIONS**: Check which HTTP methods a particular resource supports

HTTP Request/Response

Method	Request Entity-Body/Representation	Response Entity-Body/Representation
GET	(Usually) Empty Representation/entity-body sent by client	Server returns representation of resource in HTTP Response
DELETE	(Usually) Empty Representation/entity-body sent by client	Server may return entity body with status message or nothing at all
PUT	(Usually) Client’s proposed representation of resource in entity-body	Server may respond back with status message or with copy of representation or nothing at all
POST	Client’s proposed representation of resource in entity-body	Server may respond back with status message or with copy of representation or nothing at all

PUT vs. POST

- POST
  - Commonly used for creating subordinate resources existing in relation to some ‘parent’ resource
    - Parent: /weblogs/myweblog
    - Children: /weblogs/myweblog/entries/1
    - Parent: Table in DB; Child: Row in Table
- PUT
  - Usually used for modifying existing resources
  - May also be used for creating resources
- PUT vs. POST (for creation)
  - PUT: Client is in charge of deciding which URI resource should have
  - POST: Server is in charge of deciding which URI resource should have

PUT vs. POST (Cont'd)

- What in case of partial updates or appending new data? PUT or POST?
  - PUT states: Send completely new representation overwriting current one
  - POST states: Create new resource
- In practice:
  - PUT for partial updates works fine. No evidence/claim for 'why' it can't (or shouldn't) be used as such (personal preference)
  - POST may also be used and some purists prefer this

Steps to a RESTful Architecture

Read the Requirements and turn them into resources ☺

1. Figure out the data set
2. Split the data set into resources

<u>For each kind of resource:</u>

3. Name resources with URIs
4. Expose a subset of uniform interface
5. Design representation(s) accepted from client (Form-data, JSON, XML to be sent to server)
6. Design representation(s) served to client (file-format, language and/or (which) status message to be sent)
7. Consider typical course of events: sunny-day scenarios
8. Consider alternative/error conditions: rainy-day scenarios

HTTP Status/Response Codes

- HTTP is built in with a set of status codes for various types of scenarios:
  - 2xx Success (*200 OK, 201 Created*…)
  - 3xx Redirection (*303 See other*)
  - 4xx Client error (*404 Not Found*)
  - 5xx Server error (*500 Internal Server Errror*)
- Leverage existing status codes to handle sunny/rainy-day scenarios in your application!

Benefits of RESTful Design

- Simpler and intuitive design – easier navigability
- Server doesn’t have to worry about client timeout
- Clients can easily survive a server restart (state controlled by client instead of server)
- Easy distribution – since requests are independent they can be handled by different servers
- Scalability: As simple as connecting more servers ☺
- Stateless applications are easier to cache – applications can decide which response to cache without worrying about ‘state’ of a previous request
- Bookmark-able URIs/Application States
- HTTP is stateless by default – developing applications around it gets above benefits (unless you wish to break them on purpose ☺)

RESTful Frameworks

- Almost all frameworks allow you to:
  1. Specify URI Patterns for routing HTTP requests
  2. Set allowable HTTP Methods on resources
  3. Return various different representations (JSON, XML, HTML most popular)
  4. Support content negotiation
  5. Implement/follow the studied REST principles
- Jersey is ONE of the many frameworks…

List of REST Frameworks

- Rails Framework for Ruby (Ruby on Rails)
- Django (Python)
- Jersey /JAX-RS (Java)
- Restlet (Java)
- Sinatra (Ruby)
- Express.js (JavaScript/Node.js)
- …and many others: View complete list at: http://code.google.com/p/implementing-rest/wiki/RESTFrameworks

Model-View-Controller (MVC)

- Most commonly employed style with frameworks:
  - **Model**: Classes responsible for talking to the DB and fetching/populating objects for the application
  - **Controller**: Acts as URI Router i.e. routes calls to specific resources and invokes actions based on the corresponding HTTP Method
  - **View**: Usually the resource itself that returns the content/representation as requested by the client
- May/may-not be true MVC but parts of application usually split as such – leading to clean code organization/separation of concerns

Client-Side MVC

- JS heavy pages lead to spaghetti code
- Frameworks like Angular js, Backbone.js, Ember.js implement MVC paradigm on web page itself making code easier to manage/maintain
  - **Models**: Data that is fetched/saved from/to the server
  - **Views**: HTML elements that display the data and change if the data is updated
  - **Controller**: Intercepts user-events and sends appropriate messages to model/views
- JS Models communicate with server (controller) to update themselves
- Client-side MVC becoming very popular and critical for ‘front-heavy’/smart-client web-apps based on Ajax

Event-Based Architectures

- Exclusively client-side:
  - Required for communicating between various parts of the JS application/elements
  - Based on the Observer pattern – an event bus is used for sending/receiving messages across components
- Exclusively server-side:
  - For implementing asynchronous communications between different process (e.g.: sending email after a particular action)
  - Communicating with other processes on the network via a Message oriented Middleware (MoM) (e.g.: RabbitMQ, WebSphereMQ etc.)
  - Communicating with client-side apps – using Node.js or Pub/Sub web services like PubNub.com or Pusher.com

Data Access

The final nail in the coffin ☺

3-Tier Architecture

- Most commonly encountered when designing web-based systems
  - Layer 1: Presentation
    - HTML/CSS + JS (MVC)
  - Layer 2: Business Logic
    - RESTful framework (usually MVC)
  - Layer 3: Data Access
    - ORM tools – Hibernate, Spring JDBC, iBatis, Ruby's ActiveRecord & DataMapper etc.,
    - May already be integrated with RESTful framework and represented as 'Models' in the MVC

Conclusion

- Just REST isn’t enough
- 100% REST isn’t the goal either
- Various architectural styles work together in tandem for creating distributed web-based systems
- MVC on client-side is gaining high momentum
- Event-based communication exceedingly important for near-real-time/asynchronous applications (reason for Node.js popularity)
- You can learn the REST by reading a few books and designing/implementing a few systems ☺

Building REST Services with Frameworks

- REST Service can be implemented with simple servlet code at the server side

- Java provides a framework for developing REST Services
  - JAX-RS
  - Annotation-based framework
  - Provides an API specification (no official implementations)

JAX-RS Basics

An example REST service class:

```
package org.lds.tech.training.lab.ws;

import javax.ws.rs.*;

@Path("/hello")
public class HelloWebServiceRest {

    @GET
    public String sayHello() {
        return "Hello, World!";
    }
}
```

•At least one method must be annotated with an HTTP verb (e.g. @GET)

•The @Path annotation makes the class discoverable

JAX-RS Basics

An example WADL descriptor:

```xml
<application xmlns="http://wadl.dev.java.net/2009/02"
             xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <resources
      base="http://localhost:8080/example/Services/rest">
    <resource path="/">
      <method name="GET">
        <response>
          <representation mediaType="application/octet-stream">
            <param name="result" style="plain" type="xs:string"/>
          </representation>
        </response>
      </method>
    </resource>
  </resources>
</application>
```

```
@XmlRootElement(name = "employee")
@XmlAccessorType(XmlAccessType.FIELD)
public class Employee {
      private String empNo;
      private String empName;
      private String position;
}
```

```
Employee emp1 =
    new Employee("E01",
                 "Smith", "Clerk");
```

⇨

```
<employee>
   <empNo>E01</empNo>
   <empName>Smith</empName>
   <position>Clerk</position>
</employee>
```

```
List<Employee> list;
```

⇨

```
<employees>
      <employee>
         <empNo>E02</empNo>
         <empName>Allen</empName>
         <position>Salesman</position>
      </employee>
      <employee>
         <empNo>E01</empNo>
         <empName>Smith</empName>
         <position>Clerk</position>
      </employee>
      <employee>
         <empNo>E03</empNo>
         <empName>Jones</empName>
         <position>Manager</position>
      </employee>
</employees>
```

```java
List<Employee> list;
```

```json
{
    "employee": [
            {
           "empNo": "E02" ,
           "empName": "Allen",
           "position": "Salesman"
        } ,
            {
           "empNo": "E01",
           "empName": "Smith",
           "position": "Clerk"
        } ,
            {
           "empNo": "E03",
           "empName": "Jones",
           "position": "Manager"
        }
    ]
}
```

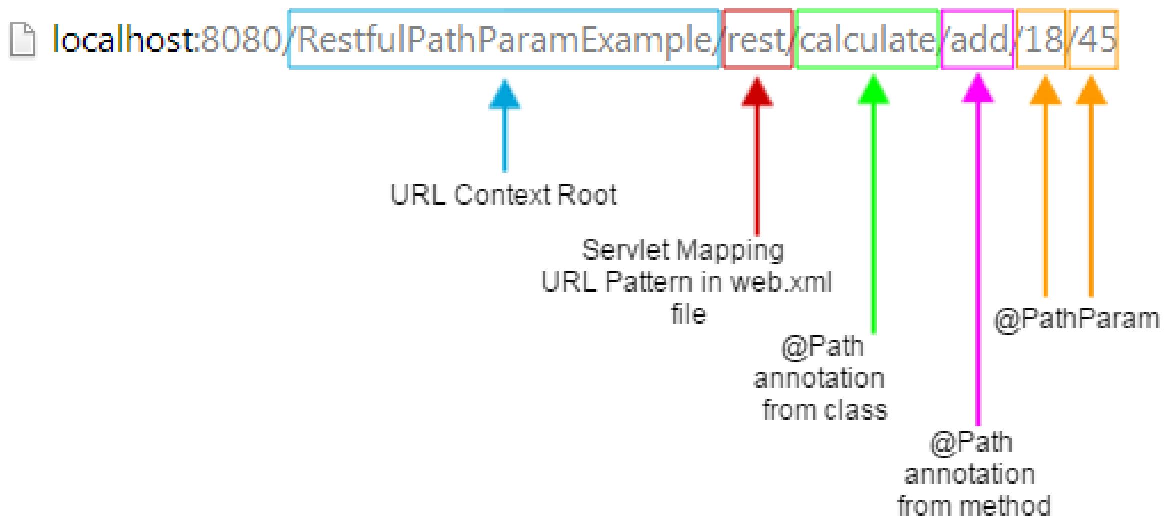
PathParam

- **@PathParam** can be used only on the following Java types:
- All primitive types except char
- All wrapper classes of primitive types except Character
- Any class with a constructor that accepts a single Stringargument
- Any class with the static method named valueOf(String) that accepts a single Stringargument
- Any class with a constructor that takes a single String as a parameter

RESTful Web Service End Points

#	URI	Method	Description
1	/rest/calculate/squareroot/{value}	GET	Calculates the square root of a number denoted by *value*
2	/rest/calculate/add/{value1}/{value2}	GET	Adds the numbers denoted by *value1* and *value2*
2	/rest/calculate/subtract/{value1}/{value2}	GET	Subtracts the numbers denoted by *value1* and *value2*

Using the @PathParam in JAX-RS API


localhost:8080/RestfulPathParamExample/rest/calculate/add/18/45
URL Context Root
Servlet Mapping URL Pattern in web.xml file
@Path annotation from class
@Path annotation from method
@PathParam

JAX-RS

- Specifications
  - @Path: the path to the resource class or method
  - @GET: the method invoked to response a GET request
  - @POST: the method invoked to response a POST request
  - @DELETE: the method invoked to response a PUT request

JAX-RS @Path Annotation

- @Path annotations may be supplied to customize the request URI of resource.

- @Path on a class defines the base relative path for all resources supplied by that class.

- @Path on a Java class method defines the relative path for the resource bound to that method.

JAX-RS @Path Annotation

- @Path on a method is relative to any @Path on the class.
- In the absence of @Path on the class or method, the resource is defined to reside at the root of the service.
- A leading forward slash (/) is not necessary as the path is always relative.

JAX-RS @Path Annotation

- @Path annotation supports the use of template parameters in the form:

{ name : regex }

```
@Path("/users/{username}")
public class UserResource {

    @GET
    @Produces("text/xml")
    public String getUser(@PathParam("username") String userName) {
        ...
    }
}
```

JAX-RS @Path Annotation

- The template parameter name is required.

- The colon (:) followed by a regular expression is optional and will default to the pattern: [^/]+

```
@Path("users/{username: [a-zA-Z][a-zA-Z_0-9]}")
```

- Multiple template parameters may be defined in a single @Path.
- Template parameter values will be injected into method parameters annotated with @PathParam.

JAX-RS Annotations: @Produces

- Used on a class or method to identify the content types that can be produced by that resource class or method.

- Method annotation overrides class annotation
- If not specified, JAX-RS assumes any type (*/*) can be produced.

JAX-RS Annotations: @Consumes

- Used on a class or method to identify the content types that can be accepted by that resource class or method.

- Method annotation overrides class annotation
- If not specified, JAX-RS assumes any type (*/*) is acceptable.
- JAX-RS responds with HTTP status “406 Not Acceptable” if no appropriate method is found.

JAX-RS Annotations

Examples of @Produces and @Consumes:

```
@Path("example")
public class ExampleRestService {

    @POST
    @Path("items")
    @Produces({"application/json", "application/xml"})
    @Consumes({"application/json", "application/xml"})
    public List<Item> editItems(List<Item> items) {
        // Does something and returns the modified list
    }
}
```

•The client *submits* JSON or XML content with the “Content-Type” header.

•The client *requests* either JSON or XML content through use of the HTTP “Accept” request header.

JAX-RS: XML and JSON Providers

- JAX-RS requires support for reading and writing XML to and from JAXB annotated classes.
- JAX-RS also requires built-in support for reading and writing JSON to and from JAXB annotated classes.
  - Default support uses Jettison as the JSON provider
  - The Stack RS namespace handler will automatically configure Jackson as the JSON provider if it is on the classpath.

JAX-RS: XML and JSON Providers

- Choosing output formats
  - passing a query parameter like format
  - specify it using extensions(changing /users url to /users.json to get the users in json format)
  - specifying the requested format(xml, json, xls, ...) by setting Accept http header.

JAX-RS: Customizing the Response

- There may be cases when you need to customize the response from your JAX-RS service:
  - To provide metadata instead of, or in addition to, the response entity.
  - To supply a custom status code
  - To instruct JAX-RS to perform a redirect
- For these cases, JAX-RS provides the abstract **Response** class and the **ResponseBuilder** utility
  - An example is provided on the following screen

JAX-RS: Customizing the Response

```java
@Path("example")
@Produces({"application/json", "application/xml"})
@Consumes({"application/json", "application/xml"})
public class ExampleRestService {

    @GET
    public List<Item> getItems() {
        // Return all items.
        return items;
    }

    @POST
    @Path("items")
    public Response editItems(List<Item> items) {
        // ... Modify the list of items
        ResponseBuilder rb = Response.temporaryRedirect(
                URI.create(UriInfo.getBaseUri() + "example"));
        return rb.build(); // redirect to getItems()
    }
}
```