RESTful Service In Java

JAX-RS

REST, What is it actually?

- REST Stands for REpresentational State Transfer
- It is an architectural style, and an approach to communications that is often used in the development and access of Web services.
- The term ***representational state transfer*** was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation at **UC Irvine**.
- Systems that conform to the constraints of REST they can be called RESTful.
- RESTful systems typically, but not always, communicate over Hypertext Transfer Protocol (HTTP)
    - They use the same HTTP verbs (GET, POST, PUT, DELETE, etc.)

REST Architectural constraints

- The formal REST constraints are
  - Client–server
  - Stateless
  - Cacheable
  - Layered system
  - Code on demand (optional)
  - Uniform interface

REST Architectural constraints – Client Server

- A uniform interface separates clients from servers.
- Clients are not concerned with data storage, which remains internal to each server, so that the portability of client code is improved.
- Servers are not concerned with the user interface or user state, so that servers can be simpler and more scalable.
- Servers and clients may also be replaced and developed independently, as long as the interface between them is not altered.

REST Architectural constraints – **Stateless**

- They are stateless
- no client context being stored on the server between requests.
- Each request from any client contains all the information necessary to service the request, and session state is held in the client.

REST Architectural constraints – **Cacheable**

- As on the World Wide Web, clients and intermediaries can cache responses.
- Responses must therefore, implicitly or explicitly, define themselves as cacheable, or not, to prevent clients from reusing stale or inappropriate data in response to further requests.
- Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance.

REST Architectural constraints – **Cacheable**

- A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way.
- Intermediary servers may improve system scalability by enabling load balancing and by providing shared caches.
- They may also enforce security policies.

REST Architectural constraints – **Code on demand (optional)**

- Servers can temporarily extend or customize the functionality of a client by the transfer of executable code.
- Examples of this may include compiled components such as Java applets and client-side scripts such as JavaScript.

REST Architectural constraints – **Uniform interface**

- Servers can temporarily extend or customize the functionality of a client by the transfer of executable code.
- Examples of this may include compiled components such as Java applets and client-side scripts such as JavaScript.

How Do RESTful applications Work?

- RESTful applications use HTTP requests to post data (create and/or update), read data (e.g., make queries), and delete data.
  - Thus, REST uses HTTP for all four CRUD (Create/Read/Update/Delete) operations.
- REST is a lightweight alternative to mechanisms like RPC (Remote Procedure Calls) and Web Services (SOAP, WSDL, et al)
  - Despite being simple, REST is fully-featured; there's basically nothing you can do in Web Services that can't be done with a RESTful architecture.
- REST is not a "standard".
  - There will never be a W3C recommendation for REST, for example.
  - And while there are REST programming frameworks, working with REST is so simple that you can often "roll your own" with standard library features in languages like Perl, Java, or C#.

REST As Web Service

- As a programming approach, REST is a lightweight alternative to Web Services and RPC.
- Much like Web Services, a REST service is:
  - Platform-independent (you don't care if the server is Unix, the client is a Mac, or anything else),
  - Language-independent (C# can talk to Java, etc.),
  - Standards-based (runs on top of HTTP), and
  - Can easily be used in the presence of firewalls.

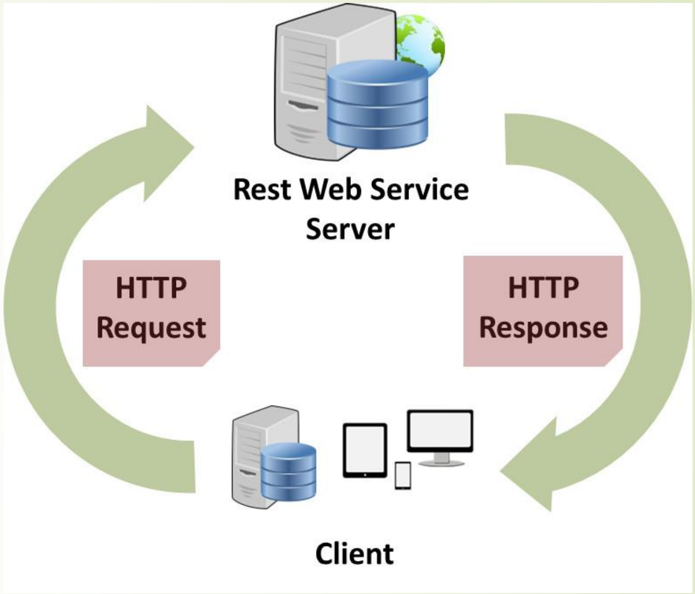
REST As Web Service

- Like Web Services, REST offers no built-in security features, encryption, session management, QoS guarantees, etc.
  - But also as with Web Services, these can be added by building on top of HTTP:
  - For security, username/password tokens are often used.
  - For encryption, REST can be used on top of HTTPS (secure sockets).

- One thing that is *not* part of a good REST design is cookies: The "ST" in "RE**ST**" stands for "State Transfer", and indeed, in a good REST design operations are self-contained, and each request carries with it (transfers) all the information (state) that the server needs in order to complete it.

REST As Web Service

- Web service APIs that adhere to the REST architectural constraints are called RESTful APIs.
- HTTP-based RESTful APIs are defined with the following aspects:
  - **base URL**, such as http://example.com/resources/
  - **an Internet media type** type that defines state transition data elements (e.g., Atom, microformats, application/vnd.collection+json) The current representation tells the client how to compose all transitions to the next application state. This could be as simple as a URL or as complex as a java applet.
  - **standard HTTP methods** (e.g., OPTIONS, GET, PUT, POST, and DELETE)



Developing RESTful Web Services using JAX-RS

What is JAX-RS?

- Java API for RESTful Web Services (**JAX-RS**) is a Java programming language API that provides support in creating web services according to the Representational State Transfer (REST) architectural pattern
- Currently JAX-RS is in release 2.x

Topics

- Goals of JAX-RS
- Address-ability
- Methods
- Representations (Formats)
- Returning status
- Statelessness
- Connectedness
- Status of JAX-RS
- Tooling

Goals of JAX-RS

Problem in Using Servlet API For Exposing a Resource (Too much coding)

```
public class Artist extends HttpServlet {
public enum SupportedOutputFormat {XML, JSON};

protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
String accept = request.getHeader("accept").toLowerCase();
String acceptableTypes[] = accept.split(",");
SupportedOutputFormat outputType = null;
for (String acceptableType : acceptableTypes) {
if (acceptableType.contains("*/*")
|| acceptableType.contains("application/*")
|| acceptableType.contains("application/xml")) {
outputType = SupportedOutputFormat.XML;
break;
} else if (acceptableType.contains("application/json")) {
outputType = SupportedOutputFormat.JSON;
break; } }
if (outputType == null)
response.sendError(415);
String path = request.getPathInfo();
String pathSegments[] = path.split("/");
String artist = pathSegments[1];
if (pathSegments.length < 2 && pathSegments.length > 3)
response.sendError(404);
/*
 * else if (pathSegments.length == 3 &&
 * pathSegments[2].equals("recordings")) { if (outputType ==
 * SupportedOutputFormat.XML) writeRecordingsForArtistAsXml(response,
 * artist); else writeRecordingsForArtistAsJson(response, artist); }
 * else { if (outputType == SupportedOutputFormat.XML)
 * writeArtistAsXml(response, artist); else writeArtistAsJson(response,
 * artist);
 */
}
}
```

There Must Be a Better Way: Server Side API Wish List for Exposing a Resource

- High level and Declarative
  - Use @ annotation in POJOs
- Clear mapping to REST concepts
  - Address-ability through URI, HTTP methods
- Takes care of the boilerplate code
  - No need to write boilerplate code
- Graceful fallback to low-level APIs when required
  - Provides ease of development with flexibility for finetuning

Address-ability

Clear mapping to REST concepts: Address-ability

- REST: A Web service exposes data as resources
  - A resource is exposed through a URI
- JAX-RS:
  - Resources are “plain old” Java classes and methods
  - The annotation **@Path** exposes a resource
  - Think resources and URIs using Java classes and **@Path**

Clear mapping to REST concepts

- Resources: what are the URIs?

  @Path("/employees/{id}")

  Variable

- Design the resource URI
  - /employees – container for 'employees'
  - /employees/123456 – one 'employee‘

- variable

  http://www.sun.com/employees/123456

  http://www.sun.com/employees/chuk

  employees id

Mapping URIs to Classes

```
@Path ("/employees")
public class Employees {
...
}
```

```
@Path ("/employees/{id}")
public class Employee {
public String getEmployee(@PathParam("id") int id) {
...
}
}
```

Methods

Clear mapping to REST concepts : Methods

- Methods: what are the HTTP methods?
- HTTP methods implemented as Java methods annotated with

  @HEAD

  @GET

  @PUT

  @DELETE

  @POST

Uniform interface: methods on root resources

```
@Path("/employees")
class Employees {
@GET <type> get() { ... }
@POST <type> create(<type>) { ... }
}
```

```
@Path(“/employees/{eid}”)
class Employee {
@GET <type> get(...) { ... }
@PUT void update(...) { ... }
@DELETE void delete(...) { ... }
}
```

Java method name is not significant
The HTTP method is the method

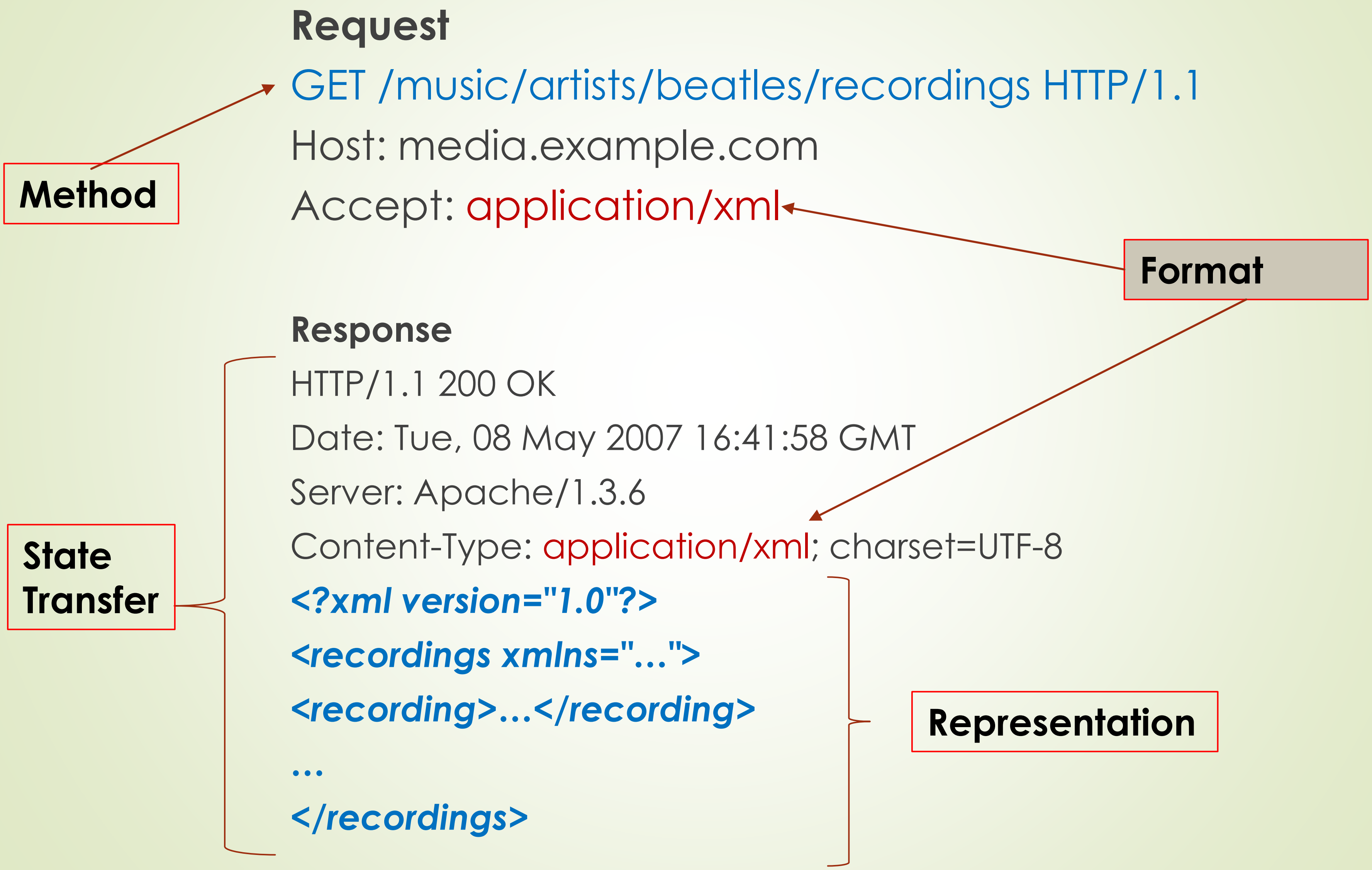
Representations (Formats)

Clear mapping to REST concepts: Formats

- **Representations:** what are the formats?

```
@Consumes("application/xml")
@Produces("application/json")
```

Formats in HTTP



Request

```
GET /music/artists/beatles/recordings HTTP/1.1
Host: media.example.com
Accept: application/xml
```

Response

```
HTTP/1.1 200 OK
Date: Tue, 08 May 2007 16:41:58 GMT
Server: Apache/1.3.6
Content-Type: application/xml; charset=UTF-8
<?xml version="1.0"?>
<recordings xmlns="…">
<recording>…</recording>
…
</recordings>
```

Multiple Representation

- Resources can have multiple representation
  - Specified through 'Content-type' HTTP header
  - Acceptable format through 'Accept' HTTP header
- A web page can be represented as
  - text/html – regular web page
  - application/xhtml+xml – in XML
  - application/rss+xml – as a RSS feed
  - application/octet-stream – an octet stream
  - application/rdf+xml – RDF format

Supported Media Types

- Think what media is consumed and produced...
- ...then think of the Java types associated
- "**Out-of-the-box**" support for the following
  - */* – byte[], InputStream, File, DataSource
  - text/* – String
  - text/xml, application/xml, – JAXBElement, Source
  - application/x-www-form-urlencoded – MultivalueMap<String, String>

Uniform Interface

Uniform interface : JAX-RS Consuming

- Specify input format with **@Consumes**
- Annotated method parameters extract client request information
  - **@QueryParam** extracts information from the URI
- Single un-annotated method parameter is the representation of the request
  - e.g. String or JAXB bean

Uniform interface: consuming

```
@Path("/employees")

class Employee{
@GET
<type> get(@QueryParam("eid") String eid)
{ }

@PUT
@Consumes("application/xml")
String update(@QueryParam("eid") String eid,Ent e)
{ ... }

@DELETE
<type> delete(@QueryParam("eid") eid) { ... }
}
```

Uniform interface : JAX-RS producing

- A HTTP method classifies the response type with

@Produces

- The method return type is the response
  - Java type that is the representation
  - Or void if no representation

Producing - annotate on methods

```
@Path("/employees")
class Employees {
@GET
@Produces("application/xml")
Col get() { ... }

@POST
@Produces("application/json")
@Produces("application/xml")
Ent create(Ent e) { ... }
}
```

Uniform interface: JAX-RS producing

- JAX-RS Service can produce an instance of **Response**
- Instance of **Response** may contain a Java type
- A response builder can produce arbitrary responses
  - e.g. created or redirected responses

Uniform interface: build a Response

```
@Path("/employees")
@Consumes("application/xml")
@Produces("application/xml")
class Employees {
@GET Col get() { ... }
@POST Response create(Ent e) {
// create and persist the new entry
// create entry resource URI
URI u = ...
// build response and return
return Response.created(u).build();
}
}
```

Returning Status

Returning Status Code

- Mainly for error conditions
- For example
  - 405 Method Not Allowed
  - 415 Unsupported Media Type
- Status can be returned either with WebApplicationException or Response

Return Code Examples

```
public class Employee {
@GET
public Employee getEmployee(@QueryParam("id") int id) {
if (!doesEmployeeExist(id))
throw new WebApplicationException(410);

}
}
```

```
public class Employee {
@GET
public Response getEmployee(@QueryParam("id") int id) {
if (!doesEmployeeExist(id))
return (Response.Builder.status(410).build());
}
}
```

Statelessness

Statelessness

- HTTP protocol is stateless
  - Service should not store session from previous requests
  - Eliminates many failure conditions
- States of Web service are resources
- Client responsible for application state
- Service responsible for resource state

Sessions are Irrelevant

- REST is the transfer of states
- Simple, visible, reusable, cacheable
- eg. Booking travel
  - Create itinerary resource, fill itinerary, post itinerary
  - All held on client as not on server session

Statelessness: JAX-RS

- Default per-request life-cycle for root resource classes
  - A new instance created for every request
  - Constructor/fields used like plain old Java objects
  - Reduces concurrency issues

Statelessness: per-request lifecycle

```
@Path("/employees/{eid}")
@Consumes("application/xml")
@Produces("application/xml")
class Employee {
String eid;
EntryResource(@QueryParam("eid") String eid)
{ this.eid = eid; }
@GET Ent get(){ ... }
@PUT void update(Ent e) { ... }
@DELETE void delete() { ... }
}
```

Statelessness: constructor can check for errors

```
@Path("/collection/{eid}")
@Consumes("application/xml")
@Produces("application/xml")
class EntryResource {
String eid;

EntryResource(@QueryParam("eid") String eid) {
this.eid = eid;
if ("eid does not exist")
// Not found
throw new WebApplicationException(404);
}
…..
}
```

JAX RS

Implementations

JAX RS Implementations

- Jersey
- JBoss RESTEasy
- Apache CXF
- Restlet
- Apache Wink

Jersey

- JAX-RS Reference Implementation: Jersey
  - Open Source
  - http://jersey.dev.java.net

Jersey Client

REST Client

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;

public class Client {
    public static void main(String[] args) {
        Client client = ClientBuilder.newClient();
        WebTarget targetUri = client
                .target("http://localhost:8080/RestService1/rest");
        String message = targetUri.path("customers").path("allemp")
                .request(MediaType.APPLICATION_JSON).get(String.class);
        System.out.println(message);
    }
}
```

Statelessness: JAX-RS guiding principles

- HTTP session life-cycle is not supported
- Developer must model state
  - As resources; and
  - As application state in the representations