

Implementing REST

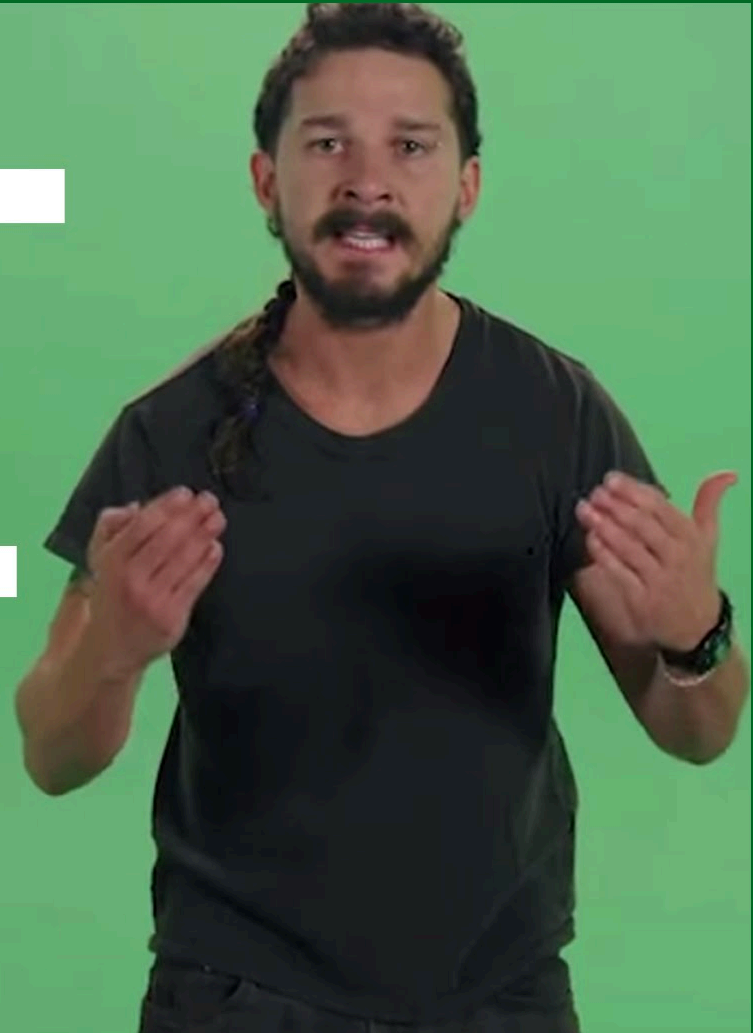
Oxford University
Software Engineering
Programme
December 2019



© Paul Fremantle 2016 except where credited elsewhere. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License
See <http://creativecommons.org/licenses/by-nc-sa/4.0/>

Just do it?!

JUST
DO IT



A good answer if you already know

- HTTP coding
- JSON
- etc



Why use a framework?

- Routing
 - Separate logic for different verbs, paths, content-types
- Cacheing and content negotiation
- Data format manipulation
 - Translation to/from JSON
- Readability



REST frameworks

Too many to list

- Java
 - JAX-RS, Spring Boot, Dropwizard, Play, WSO2 MSF4J, etc
- Node
 - Express, Restify
- .NET Web API
- Erlang
 - Leptus, WebMachine
- Python
 - Eve



Why I propose JAX-RS for Java?

- Lots of implementations is good
- Pretty decent API
 - Clean looking code
- Fast and effective



Why I use Express in node.js?

- Simple
- Good routing logic
- Works well



ymm v



© Paul Fremantle 2016 except where credited elsewhere. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License
See <http://creativecommons.org/licenses/by-nc-sa/4.0/>

Introducing JAX-RS Model

- JAX-RS uses Java annotations to map an incoming HTTP request to a Java method.
- To use JAX-RS you annotate your class with the `@Path` annotation to indicate the relative URI path.
- Then annotate one or more of your class's methods with `@GET`, `@POST`, `@PUT`, `@DELETE`, or `@HEAD` to indicate which HTTP method you want dispatched to a particular method.



An Example

```
@Path("/accounts")  
public class AccountEntryService  
{  
    @GET  
    public String getAccounts()  
{ ... }  
}
```



Query Parameters

- `getAccounts()` method could return thousands of accounts in our system.
- To limit the size of the result set, the client could send a URI query parameter to specify how many results it wanted
 - `http://somewhere.com/accounts?size=50`.
- To extract this information from the HTTP request, JAX-RS has `@QueryParam` annotation:



Accessing Query Parameters

```
@Path("/accounts")
public class AccountEntryService {
    @GET
    public String getAccounts(
        @QueryParam("size")
        @DefaultValue("50")
        int size)
    {
        ... method body ...
    }
}
```



Path Parameters

```
@Path("/accounts")
public class AccountEntryService {
    @GET
    @Path("/{id}")
    public String getAccount(
        @PathParam("id")          int
        accountId) {
        ... method body ...
    }
}
```



More on Path Parameters

- The {id} string represents our path expression.
- The @PathParam annotation will pull in the info from the incoming URI and inject it into the accountId parameter.
 - For example, if our request is <http://somewhere.com/accounts/111>, accountId would get the value 111 injected into it.
- Complex path expressions are also supported. Use Java regular expressions as follows:
 - @Path("{id: \\d+}")



Handling Content Types

- The String passed back from `getAccount()` could be any mime type: plain text, HTML, XML, JSON, YAML.
- You can specify which mime type the method return type provides with the `@Produces` annotation. For example, let's say `getAccounts()` method actually returns an XML string.
- Also the `@Consumes` can direct different incoming content types to different methods



Response Content Type

```
@Path("/accounts")
public class AccountEntryService {
    @GET
    @Path("/{id}")
    @Produces("application/xml")
    public String
    getAccount(@PathParam("id") int accountId)
    {
        ...
    }
}
```



Content Negotiation

- HTTP clients use the HTTP Accept header to specify a list of mime types they would prefer the server to return to them.
- Firefox browser sends this Accept header with every request:
 - Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;
q=0.8
- JAX-RS understands the Accept header and will use it when dispatching to JAX-RS annotated methods.



Request Content Type

```
@Path("/accounts")
public class AccountEntryService {
    @GET
    @Path("/{id}")
    @Produces("application/xml")
    public String getAccount(@PathParam("id") int
accountId) {...}

    @GET
    @Path("/{id}")
    @Produces("text/html")
    public String getAccountHtml(@PathParam("id") int
accountId) {...}
}
```



The Response Body

```
return Response.ok().  
    entity(response_body).build();
```



Content Marshalling

- JAX-RS allows you to write HTTP message body readers and writers that know how to marshall a specific Java type to and from a specific mime type.
- The JAX-RS specification has some required built-inmarshallers. For instance, vendors are required to provide support for marshalling JAXB annotated classes.
- The details are beyond this course, but look up **@Provider**



Response Codes

- The HTTP specification defines what HTTP response codes should be on a successful request.
 - GET should return 200 OK
 - POST should return 201 Created
- You can expect JAX-RS to return the same default response codes.
- Sometimes, however, you need to specify your own response codes, or simply to add specific headers or cookies to your HTTP response. JAX-RS provides a Response class for this.



Examples of creating Responses

200 OK:

```
return Response.ok().build();
```

201 Created

```
return Response.created(  
    URI.create  
    ("orders/" + uuid)).build();
```

404 Not Found

```
return  
Response.status(Status.NOT_FOUND).  
build();
```



CLIENTS

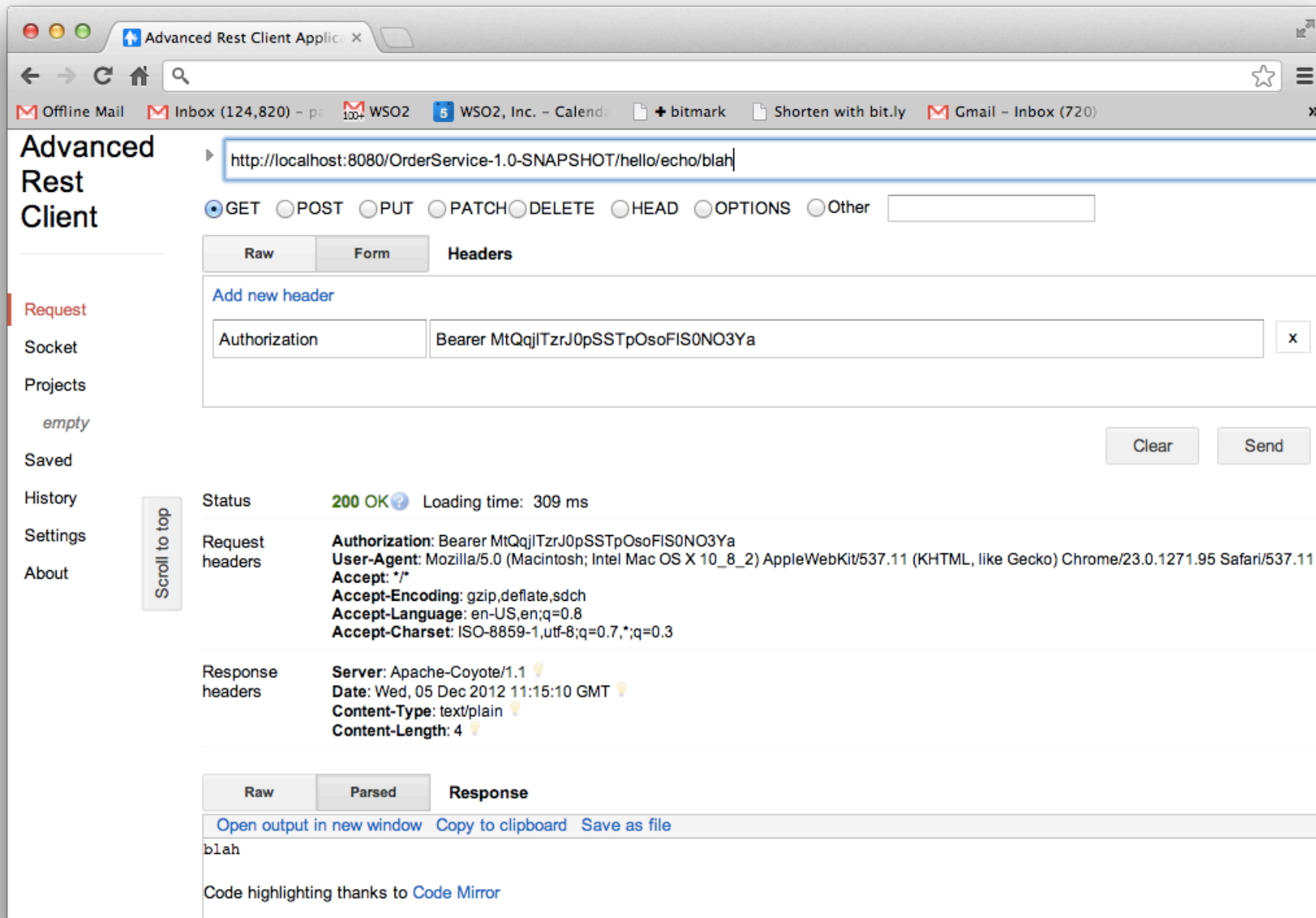


© Paul Fremantle 2016 except where credited elsewhere. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License
See <http://creativecommons.org/licenses/by-nc-sa/4.0/>

First try it out

- Chrome Advanced REST Client is a good start
- SOAPUI also provides test capabilities
- curl should be your friend too!





curl

```
curl -v http://localhost:8080/OrderService-1.0-SNAPSHOT/hello/echo/blah
* About to connect() to localhost port 8080 (#0)
*   Trying ::1...
*   connected
*   Connected to localhost (::1) port 8080 (#0)
> GET /OrderService-1.0-SNAPSHOT/hello/echo/blah HTTP/1.1
> User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0
  OpenSSL/0.9.8r zlib/1.2.5
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: Apache-Coyote/1.1
< Date: Wed, 05 Dec 2012 11:20:17 GMT
< Content-Type: text/plain
< Content-Length: 4
<
* Connection #0 to host localhost left intact
blah* Closing connection #0
```



JAXRS 2.0 Client API

- Similar to CXF client
- Aiming to be much higher level than standard HTTP clients
- Not a bad idea, but don't give up on “loose coupling”
 - The client and the service are independent
 - Technology choice of one shouldn't influence the technology choice of the other



Example JAX-RS Client Code

```
Client client =  
ClientBuilder.newBuilder().newClient();  
WebTarget target =  
client.target("http://localhost:8080/rs")  
;  
target =  
target.path("service").queryParam("a",  
"avalue");
```

```
Invocation.Builder builder =  
target.request();  
Response response = builder.get();  
Book book = builder.get(Book.class);
```



Example HttpClient code

```
HttpClient client = new DefaultHttpClient();
{
    HttpGet request = new HttpGet(url);
    HttpResponse response = client.execute(request);
    System.out.println("Response status:" + response.getStatusLine());
    System.out.println("Response data:");
    HttpEntity entity = response.getEntity();
    if (entity != null) {

        BufferedReader br = new BufferedReader(new
InputStreamReader(
            (entity.getContent())));
        String output;
        while ((output = br.readLine()) != null) {
            System.out.println(output);
        }
    }
}
```



Example Python Code

```
h = httpplib2.Http();
    resp, content =
h.request("http://api.openweathermap.org/data/2.5/weather?
q="+city)

try:
    response=json.loads(content)

    main = response['main']
    temp = round(main['temp'] - 273.15,2)
    humidity = main['humidity']
    pressure = main['pressure']
    wind = response['wind']
    windspeed = wind['speed']
    winddirection = wind['deg']
    country = response['sys']['country']
    city = response['name']
```



Example Node code

```
var post_req = http.request(post_options, function(r) {
    var body = ""
    r.on('data', function (chunk) {
        body += chunk;
    });
    r.on('end', function() {
        try {
            console.log(body);
            var response = JSON.parse(body);
        } catch (e) {}

        if (response) {
            callback(response);
        }
        else
        {
            callback(null);
        }
    });
});
```



Questions?



© Paul Fremantle 2016 except where credited elsewhere. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License
See <http://creativecommons.org/licenses/by-nc-sa/4.0/>