REST assured with JAX-RS

speak HTTP using Java

Abhishek Gupta

REST assured with JAX-RS

speak HTTP using Java

Abhishek Gupta

This book is for sale at http://leanpub.com/rest-assured-with-jaxrs

This version was published on 2016-05-03

Tweet This Book!

Please help Abhishek Gupta by spreading the word about this book on Twitter!

The suggested hashtag for this book is #RestAssuredWithJaxrs.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#RestAssuredWithJaxrs

To my wife Gitanjali and son Vihaan

Contents

Intro

This lesson is to get you started and give you an idea about JAX-RS in general. Feel free to skip this if you're not new to the framework.

What exactly is JAX-RS ?

JAX-RS is a *POJO* based, *annotation* driven framework for building web services which comply with RESTful principles. Imagine writing all the low level code to parse a HTTP request and the logic just to wire these requests to appropriate Java classes/methods. The beauty of the JAX-RS API is that it insulates the developer from the complexity and allows them to concentrate on business logic - that's precisely where the use of POJOs and annotations come into play! It has annotations to bind specific URI patterns and HTTP operations to individual methods of your Java class.

Other notable points

- More on POJO and Annotation driven approach - POJOs can be modelled as Resources with core business logic (in methods) and can be easily wired up with the HTTP request URI (/books?isbn=1234567) and HTTP verb (GET)
- Both for server and client side - The latest version of the API i.e. JAX-RS 2.0 (JSR ???) provides a client side API as well
- Standards driven - The beauty of a standards based API like JAX-RS lies in portability and vendor neutrality. One can easily switch from one implementation to another.

History

Here is a historical snapshot

Release version	Year	Java EE version	MR
JAX-RS 1.0 (JSR 311[1])	Oct 2008	Java EE 6	Nov 2009
JAX-RS 2.0 (JSR 339[2])	May 2013	Java EE 7	Sep 2014
JAX-RS 2.1 (JSR 370[3])	TBD	Java EE 8	TBD

[1] jcp.org/en/jsr/detail?id=311
[2] jcp.org/en/jsr/detail?id=339
[3] jcp.org/en/jsr/detail?id=370

- JAX-RS 2.1 is *work in progress*
- *MR* stands for Maintenance Release

Implementations

Here is a list of implementations of the JAX-RS standard

Name	Website	Latest Release (at time of writing)	Packaged with Java EE server
Jersey	jersey.java.net	2.2.22	Glassfish, Weblogic
RESTEasy	http://resteasy.jboss.org/	3.0.12	Wildfly, JBoss
Apache CXF	https://cxf.apache.org/	3.1.6	Apache TomEE

- Jersey serves as the *Reference Implementation*
- In addition to JAX-RS specification compliance, each implementation also adds it's own proprietary features

Basic tenets of RESTful services

This is the only section which talks about REST in a *conceptual* manner. Throughout the rest of the book, you'll not see references to *abstract* concepts

REST stands for Representational State Transfer. Let's quickly zip through some of the important *good-to-know-terminologies* associated with REST

- *Resources* - They are the cornerstone of REST based services. Server and client exchange *representations* of the resource. REST is not about making RPC based calls by simply using XML/JSON payloads (of course no one cab prevent you from doing so - but that cannot be called RESTful)
- *Addressable* - The resource(s) should be reachable via a URI e.g. https://api.github.com/search/repositories

- *HTTP based* - One must be able to interact with resources (CRUD) using standard HTTP (GET,PUT,POST, DELETE etc) e.g. Issue a HTTP GET to the URI https://api.github.com/search/repositories?q=
- *Stateless* - HTTP (which is the cornerstone of REST) is inherently stateless protocol and each RESTful payload should contain the related state. There is no co-relation b/w two subsequent requests even if they are the same. Also, a RESTful API should not depend on *previous* state related data to be sent by the client
- Interaction via *Hyperlinks* - Let hyperlinks drive the application. Client friendly practice popularly known as *HATEOAS* (Hypermedia As The Engine Of Application State)

What's important to know is that the JAX-RS framework embraces these principles and helps you create truly RESTful services and APIs

JAX-RS Core: Part I

The first chapter of this book covers the most fundamental concepts associated with the JAX-RS framework. It will cover the following topics

- *Resources*: foundation of RESTful applications
- *Basic annotations*: for mapping HTTP operations to your methods
- *Sub Resources*: JAX-RS takes resources one step ahead
- *Deployment specific APIs*: metadata introspected by the JAX-RS runtime during deployment

Resources

From a JAX-RS standpoint, a Resource is nothing but a class. It decorated with metadata (annotation) to further define RESTful behavior

@Path

The *@javax.ws.rs.Path* annotation defined by the JAX-RS specification is used to map the URI of the incoming HTTP request to RESTful Resources (Java classes and methods). It supports both simple strings as well as regular expressions as values

```java
@Path("/books")
public class BooksResource {
  @GET
  public Response all(){
  //send HTTP 200 as response
  return Response.ok().build();
 }
}
```

Please note - Both Java classes and methods can be annotated with @Path - The JAX-RS runtime executes a URI matching process for each incoming HTTP request which in turn takes into account @Path annotations on the class as well as method level.

Maps a HTTP GET /conferences/42 to the get() method with 42 as the input parameter

```
@Path("/conferences")
public class ConferencesResource {
  @GET
  @Path("{id}")
  public Response get(@PathParam("id") String id){
  Conference conf = getConf(id);
  return Response.ok(conf).build();
 }
}
```

We'll cover *@PathParam* (and other related annotations) in the next chapter

Mapping resource methods to HTTP verbs

Once the URI mapping is taken care of, the next logical step is to match the HTTP verb in the request to what's implemented in your service implementation.

- A specific set of JAX-RS annotations are used to map standard HTTP operations to Java methods which contain the business logic to handle the request.
- These *cannot* be applied to Java classes.
- Only one such annotation is allowed on a particular method

JAX-RS annotation	Corresponding HTTP Operation
@javax.ws.rs.GET	GET
@javax.ws.rs.PUT	PUT
@javax.ws.rs.POST	POST
@javax.ws.rs.DELETE	DELETE
@javax.ws.rs.HEAD	HEAD
@javax.ws.rs.OPTIONS	OPTIONS

Mapping HTTP methods to Java methods

```
@Path("/conferences")
public class ConferencesResource {

  @POST
  public Response create(Conference conf){
  Conference conf = createNewConf(conf);
  return Response.ok(conf.getSchedule()).build();
 }

  @DELETE
  @Path("{id}")
  public Response delete(@PathParam("id") String id){
  Conference conf = removeConf(id);
  return Response.ok().build();
 }

}
```

The above example demonstrates how different HTTP verbs (on the same URI) result in different methods being invoked on your JAX-RS resource class

- A HTTP POST (with Conference details) to /conferences results in the invocation of *create* method
- A HTTP DELETE (with Conference id as the path parameter) to /conferences/{id} results in the invocation of *delete* method

Support for 'OPTIONS'

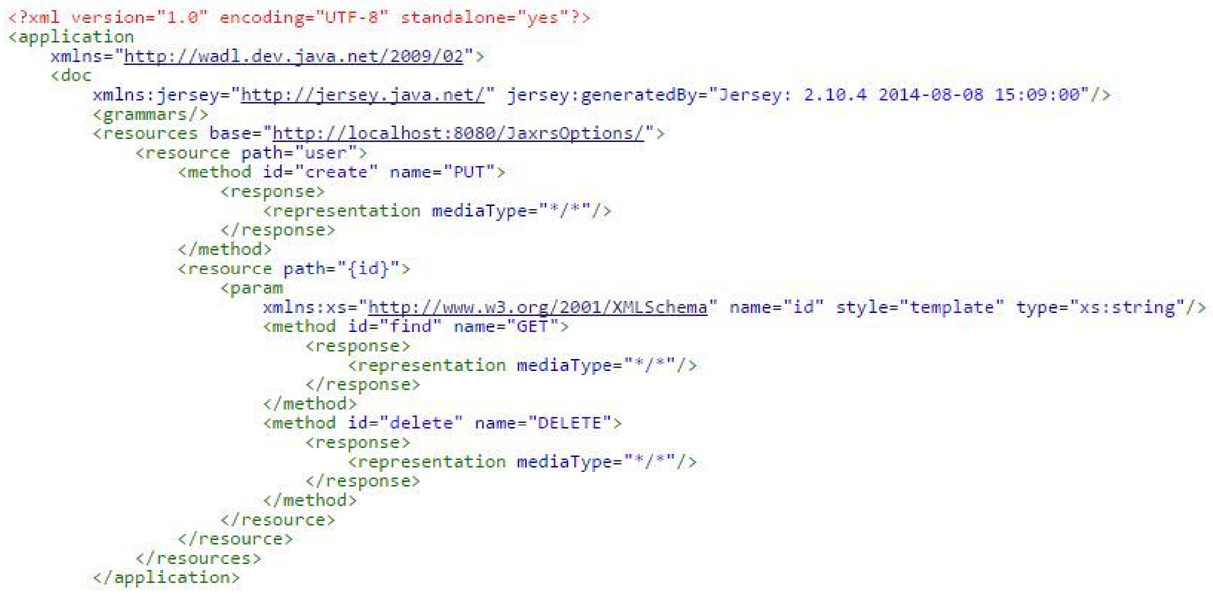
The JAX-RS specification defines sensible defaults for the HTTP OPTIONS command

typical JAX-RS resource

```
@Path("user")
public class UserResource {

    @GET
    @Path("{id}")
    public Response find(@PathParam("id") String id) {
        return Response.ok().build();
    }
```

```java

    @PUT
    public Response create(/*User user*/) {
        return Response.ok().build();
    }

    @DELETE
    @Path("{id}")
    public Response delete(@PathParam("id") String id) {
        return Response.ok().build();
    }
}
```

For the JAX-RS resource represented in the above example below, if you execute a *HTTP OPTIONS* command against the root resource URI, you will get back the WADL as a response



```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application
    xmlns="http://wadl.dev.java.net/2009/02">
    <doc
        xmlns:jersey="http://jersey.java.net/" jersey:generatedBy="Jersey: 2.10.4 2014-08-08 15:09:00"/>
        <grammars/>
        <resources base="http://localhost:8080/JaxrsOptions/">
            <resource path="user">
                <method id="create" name="PUT">
                    <response>
                        <representation mediaType="*/*"/>
                    </response>
                </method>
                <resource path="{id}">
                    <param
                        xmlns:xs="http://www.w3.org/2001/XMLSchema" name="id" style="template" type="xs:string"/>
                        <method id="find" name="GET">
                            <response>
                                <representation mediaType="*/*"/>
                            </response>
                        </method>
                        <method id="delete" name="DELETE">
                            <response>
                                <representation mediaType="*/*"/>
                            </response>
                        </method>
                    </resource>
                </resource>
            </resources>
        </application>
```

The magic of OPTIONS

To be noted

- If there is an explicit resource method which is annotated with *@javax.ws.rs.OPTIONS*, then the corresponding logic will be executed and the default behavior would get suppressed
- This does not mean that you can execute the OPTIONS command at any random URI within your RESTful app. This kicks in after the request matching process and will be applicable only for a valid URI as defined by @Path annotations in your resource classes.

Sub resource locators

Sub Resource locators are components which help you drill down to or choose from a bunch of resources based on some criteria

They enable

- resource specific logic to be implemented independently rather than mixing various permutations and combinations
- encapsulation of business logic for handing over control various resources based on criteria

A generic overview of how they work

- Use URI mapping to execute business logic to dispatch control to specific JAX-RS resource
- Once the control is transferred, the HTTP request method matching takes place and the specific method in the resource implementation class is invoked

It's important to note that sub resource locators - have the @Path annotation (to bind to correct URI) - do not contain HTTP method metadata (@GET, @POST etc.)

Sub Resource Locators in action

```
@Path("conferences")
public class ConferencesResourceLocator{

        @Path("{id}")
        public ConferenceResource pick(@PathParam("id") String confID){
                if(confID.equals("devoxx")){
                        return DevoxxResourceFactory.get(confID);
                }else if(confID.equals("JavaOne")){
                        return JavaOneResourceFactory.get(confID);
                }
        }

}
```

Devoxx specific Sub Resource

```
public class DevoxxResource extends ConferenceResource{

        @Path("{location}/{year}/{talkID}")
        @GET
        public Response get(@PathParam("year") String year,
                        @PathParam("location") String location,
                        @PathParam("talkID") String talkID){

                DevoxxConference con = getDevoxxConferenceTalkDetails(year,location,talkID);
                return Response.ok(con).build();
        }

}
```

JavaOne specific Sub Resource

```java
public class JavaOneResource extends ConferenceResource{

        @Path("{year}/{talkID}")
        @GET
        //JavaOne location is always San Francisco :-)
        public Response get(@PathParam("year") String year,
                        @PathParam("talkID") String talkID){

                JavaOneConference con = getJavaOneConferenceTalkDetails(year,talkID);
                return Response.ok(con).build();
        }

}
```

Runtime behaviour

- A HTTP GET request to /conferences/devoxx/UK/2015/42-javaeerocks
- The *ConferencesResourceLocator* forwards the request to *DevoxxResource*
- @GET annotated method (named *get*) of the the *DevoxxResource* ends up being invoked
- The *ConferencesResourceLocator* will dispatch to the *JavaOneResource* sub resource implementation in case the first URI path parameter begins with '*JavaOne*' e.g. /conferences/JavaOne/2014/42-java8rocks

Deployment specific APIs

JAX-RS applications can be deployed to a variety of environments/containers

- Java EE containers (Java EE 6 and above)
- Servlet Containers
- Java SE environment (with support from embedded HTTP container support e.g. Jetty, Tomcat etc.)

This section briefly talks about which classes and annotations are required to configure a JAX-RS powered REST service in a Java EE (*Java EE 7* to be specific) container. These API components provide the necessary information to the JAX-RS container to be able to complete the application deployment/bootstrap process

- Application
- @ApplicationPath

Application

A JAX-RS application needs to subclass this (abstract) class. It defines the components of the application and supplies additional metadata. It has a couple of method which can be overridden to return the application resource class

Abstract Method	Description
Set getClasses()	Returns a set of root resource and provider classes
Set getSingletons()	The returned set of instances of root resource and provider classes are only instantiated once

@ApplicationPath

This annotation helps define the root of your JAX-RS service i.e. it defines the base URI of the application. It can only be applied to classes which extend the *Application* class

JAX-RS configuration class

```
@ApplicationPath("rest")
public class RESTConfig extends Application{
        //empty . . . .
}
```

It's perfectly valid to have an empty implementation if custom configurations are not required

JAX-RS Core: Part II

Let's continue our coverage of JAX-RS fundamentals. In this lesson, we'll look how JAX-RS makes it possible to easily extract information from incoming client HTTP request. We specifically focus on

- HTTP *URI parameters*: path, query & matrix
- Other parts of a HTTP request: cookies, headers etc.

We use the word '*injection*' simply because it's the JAX-RS container, which does the heavy lifting of passing on the values of the individual components to our code (based on the metadata/annotations discussed in this chapter) - *inversion on control* at its best

HTTP URI parameter injection

A HTTP request URI contains: query, path and matrix parameters

- The JAX-RS API has standard annotations in order to easily inject the values of these URI parameters into your business logic
- These annotations are applicable on methods, instance variables and method parameters

JAX-RS annotation	Description	Example
@java.ws.rs.QueryParam	Injects HTTP query parameters	/talks?id=CON1234
@java.ws.rs.PathParam	Injects template parameters in URI or path segements	/users/JDOE
@java.ws.rs.MatrixParam	Injects matrix parameters	/talks/devoxx;year=2015/jvm

Parameter injection in action

```
@Path("/conferences")
public class ConferencesResource {
  @GET
  @Path("{category}")
  public Response get(@PathParam("category") String category,
                                  @QueryParam("rating") int rating){
  ConferenceSearchCriteria criteria = buildCriteria(category, rating);
  Conferences result = search(criteria);
  return Response.ok(result).build();
 }
}
```

Here is how this works

- A HTTP GET request on the URI /conferences/bigdata?rating=5
- JAX-RS container extracts the value 'bigdata' from the 'category' path parameter and injects in into your method parameter
- JAX-RS container also extracts the value '5' from the 'rating' query parameter and injects in into the method parameter
- The business logic in the method is executed, a search is performed and the applicable conferences are returned to the caller

Injecting other HTTP request components

In addition components outlined above, a HTTP request has other vital attributes which your application code might need to be aware of e.g. header variables, cookies, form data (during a POST operation) etc

These too are applicable on methods, instance variables and method parameters

JAX-RS annotation	Description
@java.ws.rs.HeaderParam	Helps capture individual HTTP headers
@java.ws.rs.CookieParam	Extracts the specified Cookie in the HTTP request
@java.ws.rs.FormParam	Pulls values of individual attributes posted via HTML forms

JAX-RS annotation	Description
@java.ws.rs.BeanParam	A convenience annotation which helps inject an instance of a class whose fields or methods are annotated with previously mentioned injection specific annotations. Prevents having to use multiple fields or method parameters to capture multiple HTTP attributes of a particular request

Injecting other stuff

```
@Path("{user}/approvals")
public class UserApprovalsResource {
  @HeaderParam("token")
  private Token authT;

  @CookieParam("temp")
  private String temp;

  @GET
  public Response all(@PathParam("user") String user){
  checkToken(user, authT);
  Logger.info("Cookie 'temp' -> "+ temp);
  Approvals result = searchApprovalsForUser(user);
  return Response.ok(result).build();
 }
}
```

The *@BeanParam* annotation was introduced in JAX-RS 2.0. Let's look at why it's so useful

@BeanParam in action

JAX-RS allows you to encapsulate the information injected via the above mentioned annotations (@PathParam, @QueryParam, @MatrixParam, @FormParam, @HeaderParam and @CookieParam) within simple POJOs.

Here is an example

@BeanParam in action I

```
public class Pagination {
  @QueryParam("s")
  private int start;
  @QueryParam("e")
  private int end;

  //getters . . .
}
```

@BeanParam in action II

```
@Path("/conferences")
public class ConferencesResource {
  //triggered by HTTP GET to /conferences?s=1&e=25
  @GET
  public Response all(@BeanParam Pagination pgSize){
  Conferences result = searchAllWithPagination(pgSize);
  //send back 25 results
  return Response.ok(result).build();
 }
}
```

Modus operandi

- Annotate the fields of the model (POJO) class with the *@Param* annotations
- Inject custom value/domain/model objects into fields or method parameters ofÂ JAX-RS resourceÂ classes using *@BeanParam*
- JAX-RS provider automatically constructs and injects an instance of your domain object which you can now use within your methods

@BeanParam is applicable to method, parameter or field

JAX-RS Core: Part III

Moving ahead, we'll look at how the JAX-RS Client API works along with the it's Security related features

- *More Injection*: what else can JAX-RS inject for free ?
- *Client API*: a standards based HTTP Client
- *Security*: authentication, authorisation, token based security etc.

Injection part II

In the previous chapter, we saw how we used specific annotations to inject HTTP URI parameters, headers, cookies etc. We'll take this one step further and see what else JAX-RS has in store for us in terms of useful *injectables* [a.k.a *freebies* ;-)]

JAX-RS provides the *@Context* annotation is used as a general purpose injection to inject a variety of resources in your RESTful services. Some of the most commonly injected components are HTTP headers, HTTP URI related information. Here is a complete list (in no specific order)

HTTP headers

Although HTTP headers can be injected using the @HeaderParam annotation, JAX-RS also provides the facility of injecting an instance of the *HttpHeaders* interface (as an instance variable or method parameter). This is useful when you want to iterate over all possible headers rather than injecting a specific header value by name

another way to work with HTTP headers

```java
@Path("testinject")
public class InjectURIDetails{
    //localhost:8080/<root-context>/testinject/httpheaders
    @GET
    @Path("httpheaders")
    public void test(@Context HttpHeaders headers){
        log(headers.getRequestHeaders().toString());
        log(headers.getHeaderString("Accept"));
        log(headers.getCookies().get("TestCookie").getValue());
    }
}
```

HTTP URI details

UriInfo is another interface whose instance can be injected by JAX-RS (as an instance variable or method parameter). Use this instance to fetch additional details related to the request URI and its parameters (query, path)

plucking out the URI information

```
@Path("testinject")
public class InjectURIDetails{
  //localhost:8080/<root-context>/testinject/uriinfo
  @GET
  @Path("uriinfo")
  public void test(@Context UriInfo uriDetails){
     log("ALL query parameters -- "+ uriDetails.getQueryParameters().toString());
     log("'id' query parameter -- "+ uriDetails.getQueryParameters.get("id"));
     log("Complete URI -- "+ uriDetails.getRequestUri());
  }
}
```

Resource Context

It can be injected to help with creation and initialization, or just initialization, of instances created by an application.

plucking out the ResourceContext

```
@Path("testinject")
public class InjectRctx{

  @Context
  ResourceContext rctx;

  //localhost:8080/<root-context>/testinject/rctx
  @GET
  @Path("rctx")
  public MyResource test(){
            //sub resource locator logic
     rctx.initResource(new MyResource());
  }
}
```

Request

Inject Request

```java
@Path("testinject")
public class InjectRequestObj{

  //localhost:8080/<root-context>/testinject/req
  @GET
  @Path("req")
  public Response test(@Context Request req){
            Date lastUpdated = …;
     req.evaluatePreConditions(lastUpdated);
     //continue further . . .
  }
}
```

More on practical usage of the Request instance in the chapter '*JAX-RS for Power Users: Part II*'

Configuration

Used to inject the configuration data associated with a *Configurable* component (e.g. Resource, Client etc.)

Retrieving Configuration data

```java
@Path("testinject")
public class InjectConfigDetails{

  @Context
  Configuration config;

  //localhost:8080/<root-context>/testinject/config
  @GET
  @Path("config")
  public void test(){
     log(config.isEnabled(MyFeature.class));
  }
}
```

Application

JAX-RS allows injection of Application subclasses as well

Injecting the Application subclass

```java
@Path("testinject")
public class InjectApplicationImpl{

  @Context
  Application theApp;

  //localhost:8080/<root-context>/testinject/app
  @GET
  @Path("app")
  public void test(){
     log(theApp.getClasses());
  }
}
```

Providers

An instance of the Providers interface can be injected using @Context. One needs to be aware of the fact that this is only valid within an existing provider. A Providers instance enables the current Provider to search for other registered providers in the current JAX-RS container

Please do not get confused between @Provider (the annotation) and Providers (the interface). More on this in the chapter *JAX-RS Providers Part III*

Security Context

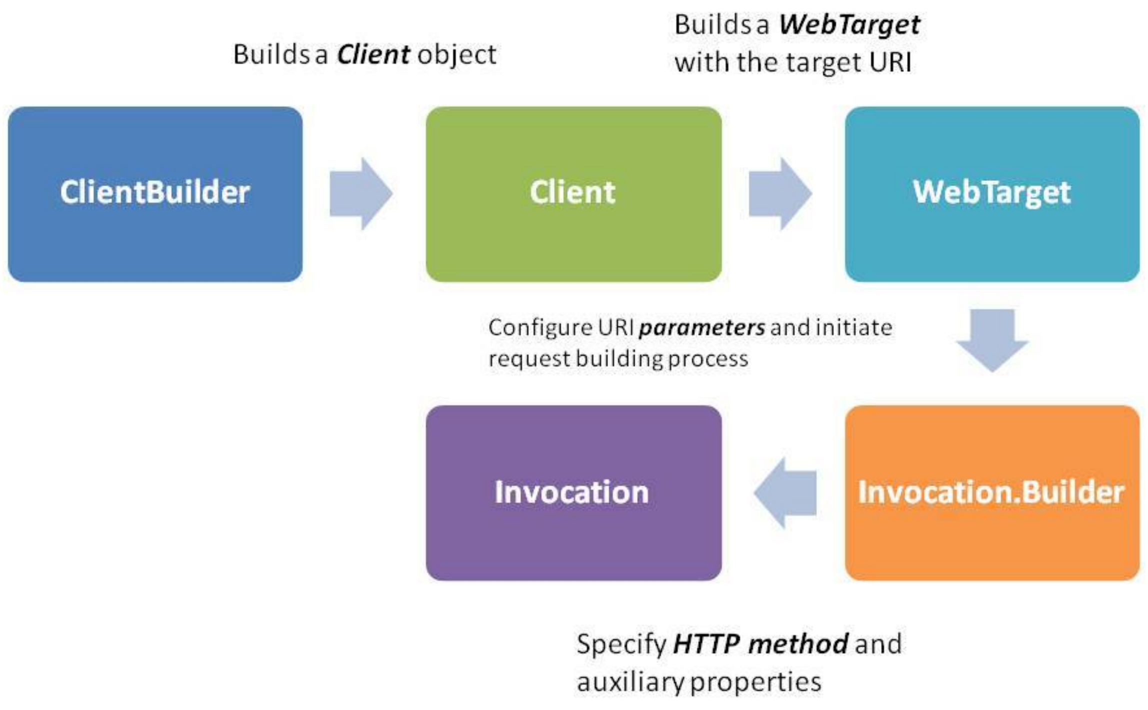
Inject an instance of the *javax.ws.rs.core.SecurityContext* interface (as an instance variable or method parameter) if you want to gain more insight into identity of the entity invoking your RESTful service.

Much more on this is the last section of this chapter

Client API

Prior to the addition of a full-fledged Client API, developers had to resort to third party implementations or interact with the HTTPUrlConnection API in the JDK to interact with HTTP oriented (REST) services. The Client API (part of *javax.ws.rs.client* package) is fairly compact, lean and fluent. It's classes and interfaces have been discussed below, followed by some code examples

A *ClientBuilder* allows you to initiate the invocation process by providing an entry point via its overloaded *newClient* methods and the build method. An instance of *Client* helps create *WebTarget* instance with the help of overloaded *target* methods. WebTarget is a representation of the URI endpoint for HTTP request invocation. It helps configure various attributes such as query, matrix and path parameters and exposes overloaded request methods to obtain an instance of Invocation.Builder. *Invocation.Builder* is responsible for further building the HTTP request and configuring attributes such as headers, cookies, cache control along with content negotiation parameters like media types, language and encoding. Finally, it helps obtain an instance of the *Invocation* object by using one of its build<HTTP_METHOD_NAME> methods. An instance of Invocation encapsulates a HTTP request and allows synchronous and asynchronous request submission via the overloaded versions of the *invoke* and *submit* method respectively.


Builds a Client object
Builds a WebTarget with the target URI
ClientBuilder
Client
WebTarget
Configure URI parameters and initiate request building process
Invocation
Invocation.Builder
Specify HTTP method and auxiliary properties

Client API

Basic example

```
Client client = ClientBuilder.newClient();
WebTarget webTarget =
client.target("https://api.github.com");
Invocation.Builder builder = webTarget
.path("search").path("repositories")
.queryParam("q", "JAX-RS")
.request("application/json");
Invocation invocation = builder.buildGet();
Response respone = invocation.invoke();
```

Let's dissect the code snippet to gain a better understanding of what's going on

- An instance of Client is obtained via ClientBuilder class
- The Client instance is used to specify the target URI as well (in this example it is https://api.github.com)
- An instance of a WebTarget is created as a result and it is further used to specify the expected response/media type (equivalent to an Accept HTTP header) and associated URI (path & query) parameters. - This creates an instance of Invocation.Builder which further builds a complete HTTP GET request-
- The Invocation instance is used to deliver the request to the server

The Configurable interface

The Client, ClientBuilder, WebTarget and Invocation objects implement the *javax.ws.rs.core.Configurable* interface. This allows them to define custom JAX-RS components such as filters, interceptors, entity providers (message readers and writers) etc. This is made possible using the overloaded versions of the *register* method [more on this later]

This is applicable to server side JAX-RS components (filters, interceptors etc) as well

Security

This section explores security aspects of the JAX-RS API

- Supported *security features* (default/out-of-the-box)
- *Configuration* style: Declarative & programmatic security

- *Bonus*: Implementing Stateless security

JAX-RS specification does not define dedicated security related features except for a few API constructs (which act as high level abstractions). For server side JAX-RS users (Java EE) it's critical to understand that the JAX-RS framework leverages the security capabilities of the container itself. To be specific, since JAX-RS is built on top the Servlet API, it has access to all the security features defined by the specification

- If you're using JAX-RS, you do not need to reinvent the wheel for securing your application
- You're free to use both declarative and programmatic security (or combination of both)
- It is flexible enough to accommodate usage of custom security frameworks if needed

Authentication & authorisation are familiar terms, so let's go over them briefly and then delve into how they can be enforced

- *Authentication*: It is the act of identification. In the context of JAX-RS, authentication involves ensuring that the caller is really who he/she/it claims to be
- *Authorization*: It is a process via which the privileges of an authenticated entity are determined. In a JAX-RS application, this would help answer

Another vital security measure, *Transport Layer Security* can be enforced using *HTTPS*

Declarative

Declarative security can be configured by using

- deployment descriptor (web.xml) or
- annotations (for role based authorization)

web.xml

web.xml is the standard deployment descriptor used by the Servlet specification. It's contained within a WAR (inside the WEB-INF folder). Amongst other parameters, it contains elements which help configuring authentication as well as role based authorization.

Let's look at a simple example

Declarative JAX-RS security

```
<web-app>
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>New Book Creation</web-resource-name>
            <url-pattern>/rest/books</url-pattern>
            <http-method>POST</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>admin</role-name>
        </auth-constraint>
        <user-data-constraint>
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
    </security-constraint>
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Book Details</web-resource-name>
            <url-pattern>/rest/books</url-pattern>
            <http-method>GET</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>*</role-name>
        </auth-constraint>
        <user-data-constraint>
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
    </security-constraint>
    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>dev-ldap</realm-name>
    </login-config>
    <security-role>
        <role-name>admin</role-name>
    </security-role>
</web-app>
```

Protected JX-RS service

```java
@Path("books")
public class BooksResource{

  //no role needed
  @GET
  @Path("{isbn}")
  public Response get(@PathParam("isbn") String isbn){
    return Response.ok(getBook(isbn)).build();
  }

  //admin role members only
  @POST
  public Response create(Book book){
    return Response.created(createBook(book).getID()).build();
  }
}
```

Implications of the above security configuration ?

- *Authentication*: Enforced using *<login-config>* element. Users will need to use their credentials and will be authenticated against the realm *dev-ldap* (imaginary LDAP directory) specified using *<realm-name>*
- *Authorisation* (role based): Any authenticated user (in any role) is allowed to fetch (using GET) book details. This is specified by the *<auth-constraint>* element. However, the book creation (using POST) service is restricted to users in *admin* role only, thanks to *<auth-constraint>* once again
- *Transport Layer* (encryption): enforced by *<user-data-constraint>*. Both GET and POST can be invoked over *HTTPS only*

Annotation based

All the annotations in this section belong to the Common Annotations specification[4]. The container/environment where they execute (in this case its the Servlet container) defines their expected behaviour and implements these annotations in a way that they are honoured at runtime.

@DeclareRoles

[4] https://jcp.org/en/jsr/detail?id=250

As the name itself indicates, this *Class* level annotation is used to declare a list of roles available within the application. Specifically, it comes into picture when programmatic (API based) authorization check is initiated by the *SecurityContext#isUserInRole(String role)* method **@RolesAllowed**

This annotation can be used on classes, individual methods or both. It specifies one or more roles which are permitted to invoke bean methods. In case the annotation is used on both class and individual methods of the bean class, the method level annotation takes precedence

@PermitAll

It can be used on both class and individual methods. If applied on a class, this annotation allows all its methods to be executed without any restrictions unless a method is explicitly annotated using *@RolesAllowed*

@DenyAll

This can be applied on a class or on specific methods. It instructs the container to forbid execution of the particular method guarded by this annotation. Please note that the method can still be used internally within the bean class

@RunAs

The use of this annotation helps impersonate a user identity (on purpose) i.e. it allows a bean method to be invoked under the context of a specific role by. This annotation can *only* be used on a class and is implicitly enforced on all the its methods

Programmatic

SecurityContext is a JAX-RS abstraction over HTTPServletRequest[5] for security related information only

It can be used for

- figuring out how the caller was *authenticated*
- extracting authenticated *Principal*[6] info
- *role* membership confirmation (programmatic authorization)
- and whether or not the request was initiated *securely* (over HTTPS)

Custom SecurityContext implementation

Custom SecurityContext ? Why?

It helps when you have a custom authentication mechanism not implemented using standard Java EE security realm. A typical example is token based authentication based on custom (app specific) HTTP headers

[5]http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServletRequest.html

[6]https://docs.oracle.com/javase/8/docs/api/java/security/Principal.html

- the web container is not be aware of the authentication detail. Hence, the SecurityContext instance will *not* contain the subject, role and other details
- the JAX-RS request pipeline needs to be aware of the associated 'security context' & make use of it within its business logic

SecurityContext is an interface after all. All you need to do is just implement, inject (using @Context[7]) and use it !

Custom SecurityContext implementation

```java
@Priority(Priorities.AUTHENTICATION)
public class AuthFilterWithCustomSecurityContext implements ContainerRequestFilt\
er {
    @Context
    UriInfo uriInfo;

    @Override
    public void filter(ContainerRequestContext requestContext) throws IOExceptio\
n {
        String authHeaderVal = requestContext.getHeaderString("Auth-Token");
        //execute custom authentication
        String subject = validateToken(authHeaderVal);
        if (subject!=null) {
            final SecurityContext securityContext = requestContext
                                .getSecurityContext();
            requestContext.setSecurityContext(new SecurityContext() {
                        @Override
                        public Principal getUserPrincipal() {
                            return new Principal() {
                                @Override
                                public String getName() {
                                    return subject;
                                }
                            };
                        }

                        @Override
                        public boolean isUserInRole(String role) {
                            List<Role> roles = findUserRoles(subject);
                            return roles.contains(role);
                        }
```

[7] https://docs.oracle.com/javaee/7/api/javax/ws/rs/core/Context.html

```

                        @Override
                        public boolean isSecure() {
                            return uriInfo.getAbsolutePath().toString()
                                                    .startsWith("https");
                        }

                        @Override
                        public String getAuthenticationScheme() {
                            return "Token-Based-Auth-Scheme";
                        }
                    });
        }

    }
}
```

Stateless ‘token’ based security

This section discusses

- Provides a quick intro to JWT
- Shows how to use it with JAX-RS (for authentication) with an example[8]

The **jose4j** library[9] was used for JWT creation and validation

Brief intro to JWT

- A standard defined by RFC 7519[10]
- Used to exchange *claims*
- Has a pre-defined *structure*

[8]https://github.com/abhirockzz/jaxrs-with-jwt
[9]https://bitbucket.org/b_c/jose4j/wiki/Home
[10]https://tools.ietf.org/html/rfc7519

Anatomy of a JWT

It consists of *three* parts

- *Header*: consists of info like signature mechanism, token type etc.
- *Body (Claims)*: the meat of the payload
- *Signature*: signature of the contents to protect against tampered/malicious JWTs

These three components come together to form the actual token

JWT building blocks

```
//header

{
  "alg": "HS256",
  "typ": "JWT"
}

//payload/claims

{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}

//the formula

encoded_part = base64Of(header) + "." base64Of(payload)
//assume that algo is HS256 and secret key is 'secret'

signature = signedUsingHS256WithSecret(encoded_part)
JWT = encoded_part + "." + sigature

//the JWT ( notice the separator/period --> "." )

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9 //base-64 encoded header
.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9 //base64 e\
ncoded payload
.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ //the signature
```

Benefits of using JWT

- Useful for implementing *Stateless* authentication
- *Compact*: less verbose compared to other counterparts like SAML)
- *Flexible*: Although its backed by a standard, you are free to choose your signature, claim attributes etc.

Please note that…

- JWT is *not only* an authentication mechanism. It's more about information exchange & it's usage is limited by your imagination
- It's signed, *not encrypted*: its contents can be picked up over the wire if you do not secure your transport layer (e.g. using HTTPS)

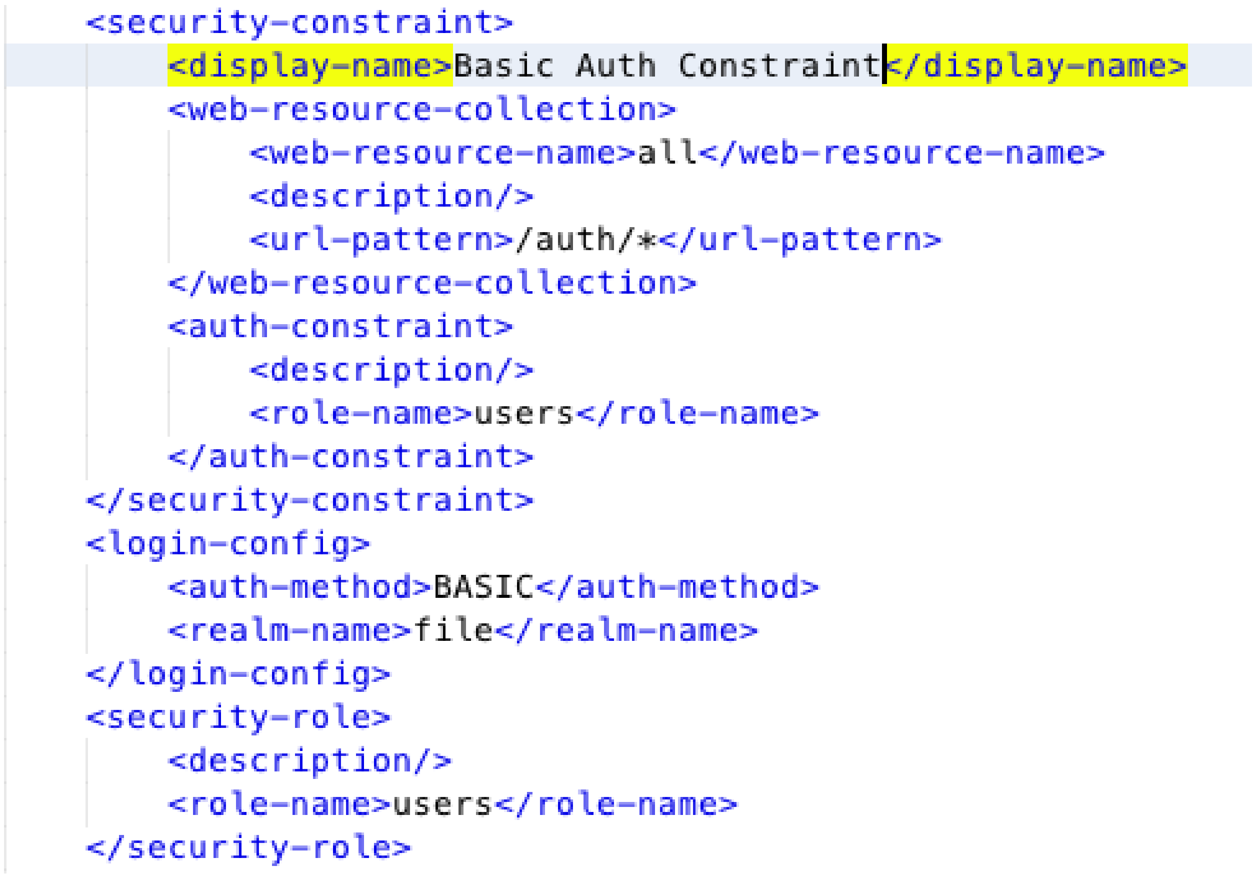
Using JWT with JAX-RS

Let's look at an example of how we might use JWT in a JAX-RS based application. As stated earlier, this sample uses JWT as a stateless authentication token. The process is split into distinct steps

Getting hold of the JWT

Why do we need a JWT in the first place? *It is because the JAX-RS resource is protected and its access is dependent on the presence of a JWT token within the HTTP request (this is achieved by a JAX-RS filter)* Think of JWT as a *proxy* to the actual username/password (or any other authentication criteria) for your application. You need to actually authenticate using the method required by your application in order to get access to the JWT. In this example, a successfully executed HTTP Basic authentication is the gateway to the token This is what happens

1. The application executes a GET request to the URL http://<host>:<port>/<context-root>/auth/token with the HTTP *Authorization* header containing user credentials
2. HTTP Basic authentication kicks in. This is enforced by the *web.xml* (snippet below) which ensures that any request to the /auth/* is not allowed to pass unauthenticated
3. In case of a successful authentication, the *JWT is returned* in the HTTP response header



```
    <security-constraint>
        <display-name>Basic Auth Constraint</display-name>
        <web-resource-collection>
            <web-resource-name>all</web-resource-name>
            <description/>
            <url-pattern>/auth/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <description/>
            <role-name>users</role-name>
        </auth-constraint>
    </security-constraint>
    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>file</realm-name>
    </login-config>
    <security-role>
        <description/>
        <role-name>users</role-name>
    </security-role>
```

Security configuration in web.xml deployment descriptor

JWT creation

```
//exception handling excluded to avoid verbosity

RsaJsonWebKey rsaJsonWebKey = RsaKeyProducer.produce();

JwtClaims claims = new JwtClaims();
claims.setSubject("user1");

JsonWebSignature jws = new JsonWebSignature();
jws.setPayload(claims.toJson());
jws.setKey(rsaJsonWebKey.getPrivateKey());
jws.setAlgorithmHeaderValue(AlgorithmIdentifiers.RSA_USING_SHA256);

String jwt = jws.getCompactSerialization();

//the encoded JWT

```

```
eyJhbGciOiJSUzI1NiJ9
.eyJzdWIiOiJ1c2VyMSJ9
.HG9GCQPuC6w6pulbYE2uurCzpEwoWvz_8Ps5ZjgtfomyY4LWacDEzlHLnyMj9H7aqgcePC7_4l2wDXQ\
V-S0BQRsIZfJeUUmWxlTlLzvKZr_2eEx00YZPPFZNoFCfwB-ajLHLLenROy4aSjPo_Vg9o7N-p0DZ1yZ\
QoJhkvoVJgkhX9FeAf65kIZkbuJC9dmVkzXSOpVf4GZeCpNDJJYSo6IAnL3UEoWek6V9BtWgV-a4xvyd\
p7vxkdDXmzmalGLYuWbuVG7rWcbWwSfsg38iEG-mqptqA_Kzk1VmjwWNo_BfvLuzjzuosqi732-5SRzB\
P-2zqGghBqMYsGgkqkH2n7A

//human readable format

{
  "alg": "RS256" //header
}

{
  "sub": "user1" //claim payload
}
```

Leveraging the JWT

- The JWT is sent by app in the subsequent request for the JAX-RS resource i.e.http:<host>:<port>/<context-root>/resources/books
- The JAX-RS *Container Request Filter* kicks in - it checks for the presence of the JWT , verifies it. The verification process implicitly checks for *presence of the required claim attributes* as well as the *signature validation*

Extracting JWT from HTTP header

```
@Priority(Priorities.AUTHENTICATION)
public class JWTAuthFilter implements ContainerRequestFilter{
  @Override
  public void filter(ContainerRequestContext requestContext) throws IOException {
        String authHeaderVal = requestContext.getHeaderString("Authorization");

            //consume JWT i.e. execute signature validation
            if(authHeaderVal.startsWith("Bearer")){
            try {
                validate(authHeaderVal.split(" ")[1]);
            } catch (InvalidJwtException ex) {
                requestContext
                .abortWith(Response.status(Response.Status.UNAUTHORIZED).build()\
);
```

```
            }
            }else{
                requestContext
                .abortWith(Response.status(Response.Status.UNAUTHORIZED).build()\
);
            }
  }
```

JWT verification

```
//should be the same as the one used to build the JWT previously
RsaJsonWebKey rsaJsonWebKey = getCachedRSAKey();

JwtConsumer jwtConsumer = new JwtConsumerBuilder()
                // the JWT must have a subject claim
      .setRequireSubject()
      // verify the signature with the public key
       .setVerificationKey(rsaJsonWebKey.getKey())
        .build(); // create the JwtConsumer instance

JwtClaims jwtClaims = jwtConsumer.processToClaims(jwt);
```

- It allows the request to go through in case of successful verification, otherwise, the filter returns a *HTTP 401 Unauthorized* response to the client
- A *container response filter* ensures that the JWT is added as a part of the response header again. It only does so when the JWT verification was successful - this is made possible using the contextual state/information sharing feature provided by JAX-RS Request Filters

Response filter makes use of the JWT validation result

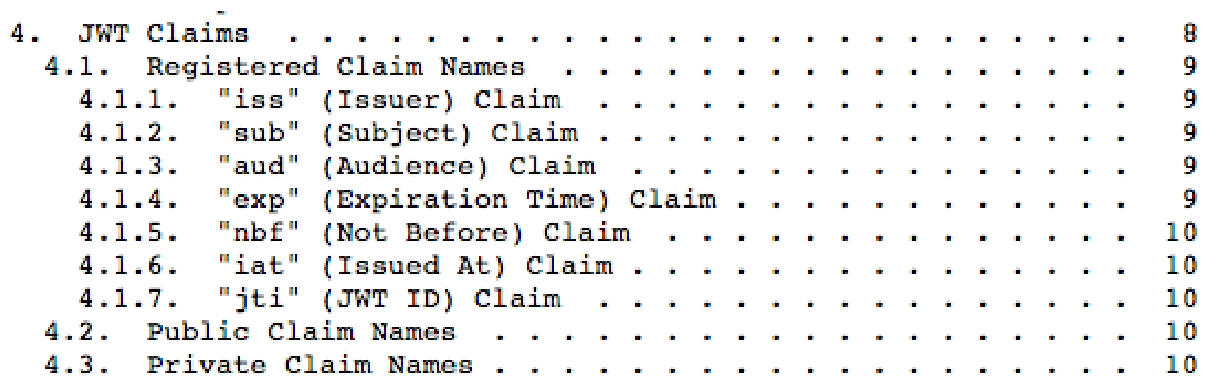
```
public class JWTResponseFilter implements ContainerResponseFilter {

    @Override
    public void filter(ContainerRequestContext requestContext, ContainerResponse\
Context responseContext) throws IOException {
        System.out.println("response filter invoked...");
        if (requestContext.getProperty("auth-failed") != null) {
            Boolean failed = (Boolean) requestContext.getProperty("auth-failed");
            if (failed) {
                System.out.println("JWT auth failed. No need to return JWT token\
");
```

```
                return;
            }
        }

        List<Object> jwt = new ArrayList<Object>();
        jwt.add(requestContext.getHeaderString("Authorization").split(" ")[1]);
        responseContext.getHeaders().put("jwt", jwt);
        System.out.println("Added JWT to response header 'jwt'");

    }
}
```

Other considerations

- **Choice of claim attributes**

In this example, we just used the standard sub (subject) attribute in the claim. You are free to use others. I would highly recommend reading section 4 of the JWT RFC for deeper insight



```
4.  JWT Claims . . . . . . . . . . . . . . . . . . . . . . . .   8
  4.1.  Registered Claim Names  . . . . . . . . . . . . . . . .   9
    4.1.1.  "iss" (Issuer) Claim  . . . . . . . . . . . . . . .   9
    4.1.2.  "sub" (Subject) Claim . . . . . . . . . . . . . . .   9
    4.1.3.  "aud" (Audience) Claim  . . . . . . . . . . . . . .   9
    4.1.4.  "exp" (Expiration Time) Claim . . . . . . . . . . .   9
    4.1.5.  "nbf" (Not Before) Claim  . . . . . . . . . . . . .  10
    4.1.6.  "iat" (Issued At) Claim . . . . . . . . . . . . . .  10
    4.1.7.  "jti" (JWT ID) Claim  . . . . . . . . . . . . . . .  10
  4.2.  Public Claim Names  . . . . . . . . . . . . . . . . . .  10
  4.3.  Private Claim Names . . . . . . . . . . . . . . . . . .  10
```

Refer to the RFC for a deep dive on JWT claimsx

- **JWT expiration**

One should also consider expiring the JWT token after a finite time. You would need to

- Make use of the *exp* claim attribute (standard)
- Think about refreshing the JWT token (after expiry)

Revisiting the Stateless paradigm Although the initial authentication was executed using HTTP Basic, the application does not rely on a Session ID for authorising subsequent requests from the same user. This has the following implications

- There is no need to *store* the session ID on the server side
- There is no need to *sync* this session ID to multiple application nodes in a cluster

As stated above, JWT is helping us with Stateless authentication (it is not very different from the HTTP protocol itself)

- Our JWT contains all the required data (claim) for the conversation (in this case authentication)
- We pass the token with each HTTP request (only to access resources which are protected by the JWT to begin with)
- The application does not need to repetitively authenticate the user (via the username-password combo)

Now we can scale :-)

You can have multiple instances (*horizontally scaled* across various nodes/clusters) of your JAX-RS service and yet you need not sync the state of the token between various nodes. If a subsequent request goes to different node than the previous request, the authentication will still happen (provided you pass the JWT token)

JAX-RS Providers: Part I

Providers ???

They are nothing but implementations of specific JAX-RS interfaces which provide flexibility and extensibility. They either need to be annotated with the @Provider annotation for automatic detection by the JAX-RS container or need to explicitly configured

In this lesson, we are going to cover two providers

- *Message Body Reader*: HTTP payload to Java object transformers
- *Message Body Writer*: convert Java object into HTTP payloads before sending them to the caller
- *JAX-RS support for JSON-P artefacts*

Message Body Reader

A *MessageBodyReader* is an interface whose implementation supports the conversion of HTTP request content to a Java type which can then be consumed by your JAX-RS application.

Default support

Every JAX-RS implementation provides out-of-the-box support for existing data types (i.e. a default Message Body Reader is provided) such as String, primitives, InputStream, Reader, File, byte array (byte[]), JAXB annotated classes, JSON-P objects. The same is applicable for Message Body Writers as well (see the next topic)

It's best to understand this with the help of an example

Message Body Reader implmentation

```
@Provider
@Consumes(MediaType.APPLICATION_XML)
public class CustomerDataToCustomerJPAEntity
        implements MessageBodyReader<CustomerJPA> {


    //implementation for MessageBodyReader interface

    @Override
    public boolean isReadable(Class<?> type, Type type1, Annotation[] antns, Med\
iaType mt) {

        //return true unconditionally - not ideal. one can include checks
        return true;
    }

    @Override
    public LegacyPOJO readFrom(Class<LegacyPOJO> type, Type type1, Annotation[] \
antns, MediaType mt, MultivaluedMap<String, String> mm, InputStream in) throws I\
OException, WebApplicationException {

        JAXBContext context = null;
        Unmarshaller toJava = null;
        CustomerJAXB jaxbCust = null;
        CustomerJPA jpaCust = null;

        try {
            // unmarshall from XML/JSON to Java object

             context = JAXBContext.newInstance(CustomerJAXB);
             toJava = context.createUnmarshaller();
             jaxbCust = (CustomDomainObj) toJavaObj.unmarshal(in);

             //build JPA entity
                    jpaCust = new CustomerJPA(jaxbCust.getUniqueID(), jaxbCust.getName(\
));
        } catch (JAXBException ex) {
            Logger.getLogger(MessageTransformer.class.getName()).log(Level.SEVER\
E, null, ex);
        }
        return jpaCust;
    }
}
```

Message Body Reader in action

```
@Stateless
@Path("customers")
public class CustomersResource{

        @PersistenceContext
        EntityManager em;

         @POST
         @Consumes(MediaType.APPLICATION_XML)
         //XML payload -> JAXB -> JPA
    public Response create(CustomerJPA cust){
             //create customer in DB
             em.persist(cust);
             //return the ID of the new customer
        return Response.created(cust.getID()).build();
    }
}
```

What's going on here ?

- client sends XML payload (in HTTP message body representation)
- JAX-RS scans the available Message Body Readers and finds our implementation - *CustomerDataToCustomerJPAEntity*
- As per our requirement, we first transform the raw payload into an instance of our JAXB annotated model class (*CustomerJAXB*) and then build an instance of our custom JPA entity (*CustomerJPA*)

Benefits

- Separates the business logic from data transformation code - the conversion is transparent to the application
- One can have different implementations for conversion of different on-wire representations to their Java types and qualify them at runtime by specifying the media type in the @Produces annotation e.g. you can have separate reader implementations for a GZIP and a serialised (binary) representation to convert them to the same Java type

Message Body Writer

Now that we have seen Readers, Message Body Writers are easy to understand - they are the exact opposite i.e. an implementation of a Message Body Writer transforms a Java type to an on-wire format to be returned to the client.

The below example, is the exact opposite (mirror image) of what was demonstrated earlier. This time, we are returning an instance of our custom class without including any transformation logic in our JAX-RS resource classes - it's encapsulated within our MessageBodyWriter implementation

Message Body Writer implmentation

```
@Provider
@Produces(MediaType.APPLICATION_XML)
public class CustomerJPAEntityToCustomerData
        implements MessageBodyWriter<CustomerJPA> {

        @Override
    public boolean isWriteable(Class<?> type, Type type1, Annotation[] antns, Me\
diaType mt) {
       //return true unconditionally - not ideal. one can include checks
        return true;
    }

    @Override
    public long getSize(CustomerJPA t, Class<?> type, Type type1, Annotation[] a\
ntns, MediaType mt) {
         return -1;
    }

    @Override
    public void writeTo(CustomerJPA t, Class<?> type, Type type1, Annotation[] a\
ntns, MediaType mt, MultivaluedMap<String, Object> mm, OutputStream out)
    throws IOException, WebApplicationException {

        JAXBContext context = null;
        Marshaller toXML = null;
        CustomerJAXB jaxbCust = new CustomerJAXB();

        jaxbCust(t.getId());
        jaxbCust(t.getName());

        try {
            // marshall from XML/JSON to Java object

             context = JAXBContext.newInstance(CustomerJAXB.class);
             toXML = context.createMarshaller();

             //write marshalled content back to client
             toXML.marshal(jaxbCust, out);

        } catch (JAXBException ex) {
            Logger.getLogger(MessageTransformer.class.getName()).log(Level.SEVER\
E, null, ex);
        }
    }

}
```

Message Body Writer in action

```
@Stateless
@Path("customers")
public class CustomersResource{

        @PersistenceContext
        EntityManager em;

         @GET
         @Produces(MediaType.APPLICATION_XML)
         @Path("{id}")
         //JPA -> JAXB -> XML payload
    public Response get(@PathParam("id") String custID){
             //search customer in DB
             CustomerJPA found = em.find(CustomerJPA.class,custID);
       return Response.ok(found).build();
    }
}
```

JAX-RS and JSON-P integration

This section talks about support for JSON-P in JAX-RS 2.0

JSON-P ...?

The JSON Processing API[11] (JSON-P) was introduced in Java EE 7[12]. It provides a standard API to work with JSON data and is quite similar to its XML counterpart - JAXP[13]. JSON-B[14] (JSON Binding) API is in the works for Java EE 8[15].

Support for JSON-P in JAX-RS 2.0

JAX-RS 2.0[16] (also a part of Java EE 7) has out-of-the-box support for JSON-P artifacts like JsonObject[17], JsonArray [18]and JsonStructure [19]i.e. every JAX-RS 2.0 compliant implementation will provide built in Entity Providers for these objects, making it seamless and easy to exchange JSON data in JAX-RS applications

Let's look at a few code samples

Returning JSON array

```
@GET
public JsonArray buildJsonArray(){
  return Json.createArrayBuilder().add("jsonp").add("jaxrs").build();
}
```

Accepting JSON Object as payload

```
@POST
public void acceptJsonObject(JsonObject payload){
  System.out.println("the payload -- "+ payload.toString());
}
```

These are pretty simple examples, but I hope you get the idea....

[11]http://jcp.org/en/jsr/detail?id=353
[12]http://www.jcp.org/en/jsr/detail?id=342
[13]https://www.jcp.org/en/jsr/detail/summary?id=206
[14]https://www.jcp.org/en/jsr/detail?id=367
[15]https://www.jcp.org/en/jsr/detail?id=366
[16]https://www.jcp.org/en/jsr/detail?id=339
[17]https://docs.oracle.com/javaee/7/api/javax/json/JsonObject.html
[18]https://docs.oracle.com/javaee/7/api/javax/json/JsonArray.html
[19]https://docs.oracle.com/javaee/7/api/javax/json/JsonStructure.html

Few things to be noted

- No need to write custom MessageBodyReader or MessageBodyWriter implementations. As mentioned previously, the JAX-RS implementation does it for you for free
- This feature is *not the same* as being able to use JAXB annotations on POJOs and exchange JSON versions of the payload (by specifying the application/xml media type) - this is also known as *JSON binding* and it is one of the potential candidates for* Java EE 8*. I have experimented with this and observed that GlassFish 4.1 (Jersey) and Wildfly 8.x (RESTEasy) support this by default

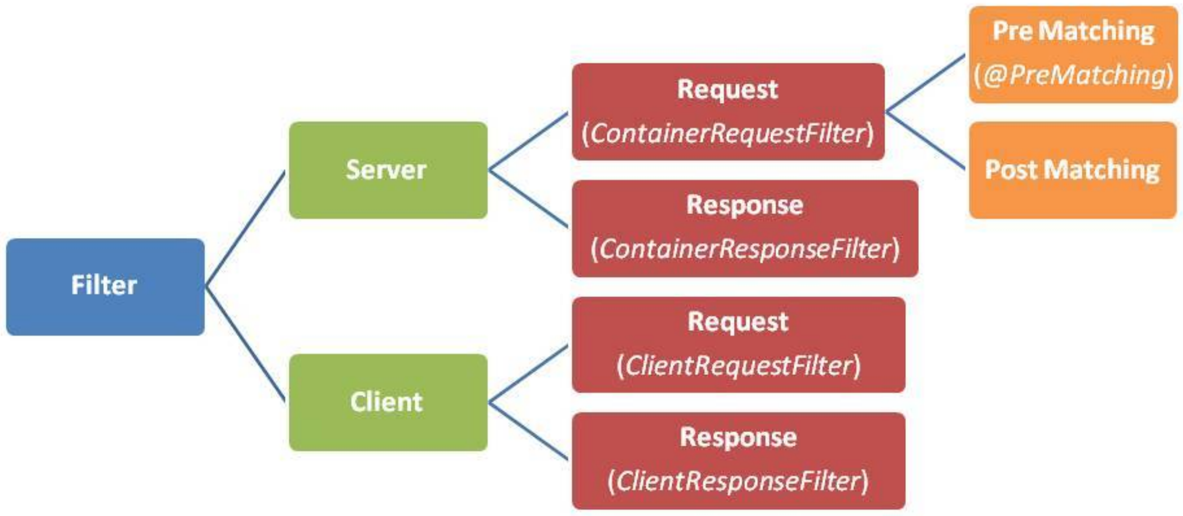
JAX-RS Providers: Part II

Let's continue exploring different JAX-RS providers and dive into

- *Filters*: one of the foremost components of the JAX-RS request processing chain
- *Interceptors*: work in tandem with (intercept) Message Body Readers and Writers

Filters

Filters provide *AOP* (Aspect Oriented Programming) like capabilities within JAX-RS applications and allow developers to implement cross cutting application specific concerns which ideally should not be sprinkled all over the business logic e.g. authentication, authorization, request/response validation etc. The AOP-based programming model involves interposing on methods of JAX-RS resource classes and dealing with (or mutating) components of HTTP request/response - headers, request URIs, the invoked HTTP method (GET, POST etc)



JAX-RS filters

Server side Request filter

Server side request filters act on incoming HTTP requests from the clients and provide an opportunity to act on/make decisions based on certain characteristics of the HTTP request. In order to implement a server side request filter, one needs to implement the *javax.ws.rs.container.ContainerRequestFilter* interface (which is an extension provided by JAX-RS). An instance of the *javax.rs.ws.ContainerRequestContext* interface is seamlessly injected by the container into the filter method of ContainerRequestFilter. It is a mutable object (on purpose) and exposes methods to access and modify HTTP request components

A JAX-RS request processing pipeline involves dispatching a HTTP request to the appropriate Java method in the resource classes based on matching algorithm implemented by the JAX-RS provider. Filters take this into account and are divided into *pre* and *post* matching filters

Pre-matching filters

As the name indicates, Pre-matching filters are executed before the incoming HTTP request is mapped/dispatched to a Java method. Use the *javax.ws.rs.container.PreMatching* annotation on the filter implementation class

Server side Pre Matching filter

```java
@Provider
@PreMatching
public class PreMatchingAuthFilter{
 public void filter(ContainerRequestContext crc)
 throws IOException{
  if(crc.getHeaderString("Authorization") == null){
   crc.abortWith(Response.status(403).build());
   }else{ //check credentials.... }
  }
}
```

Post-matching filters

A Post-matching filter is executed by the JAX-RS container only after the completion of method dispatch/matching process. Unlike, pre-matching filters, these filters do not need an explicit annotation i.e. filter classes without the @PreMatching annotation are assumed to be post-matching by default

Server side Post Matching filter

```java
@Provider
public class PostMatchingFilterExample{
 public void filter(ContainerRequestContext crc)
 throws IOException{
  System.out.println("Referrer: "+
  crc.getHeaderString("referrer"));
  System.out.println("Base URI: "+
  crc.getUriInfo().getBaseUri());
  System.out.println("HTTP Request method: "+
  crc.getMethod());
  }
}
```

A JAX-RS application can have multiple filters (in a *chain* like structure) which are executed as per user defined order (more on this later) or a default one (container driven). However, it possible to break the chain of processing by throwing an exception from the filter implementation logic or by calling the *abortWith* method. In either cases, the other request filters in the chain are not invoked and the control is passed on to the Response Filters (if any).

Server side Response filter

A server side Response filter is invoked by the runtime after a response (or an exception) is generated by the JAX-RS resource method (before dispatching the same to the caller/client). Response filters are similar to their counterparts (Request filters) in terms of their utility (read/mutate aspects of the response e.g. HTTP headers) and programming model (executed as a chain in a user defined or default order) Setting up a server side response filter is as simple as providing an implementation for the *javax.ws.rs.container.ContainerResponseFilter* interface. The injection of *ContainerResponseContext* into the filter method of the ContainerResponseFilter interface is taken care of by the JAX-RS runtime

Response filters also need to be annotated with the *javax.ws.rs.ext.Provider* annotation in order for the JAX-RS runtime to recognize it automatically.

Server side Response filter

```
@Provider
public class CustomerHeaderResponseFilter{
 public void filter(ContainerRequestContext crc,
 ContainerResponseContext resCtx)
 throws IOException{
  System.out.println("Request URI: "+
  crc.getUriInfo().getAbsolutePath().toString());
  //adding a custom header to the response
  resCtx.getHeaders().add("X-Search-ID",
  "qwer1234-tyuio5678-asdfg9876");
  }
}
```

Client side Request filter

Client side request filters are invoked after the HTTP request creation before it is dispatched to the server. They can be to mutate/make decisions based on the properties of the HTTP request (Headers, Cookies etc) In order to implement a client side request filter, one needs to implement the extension interface provided by JAX-RS - *javax.ws.rs.client.ClientRequestFilter*

Client side Request filter

```
public class ClientRequestHTTPMethodFilter {
 public void filter(ClientRequestContext crc)
 throws IOException{
  String method = crc.getMethod();
  if(method.equalsIgnoreCase("DELETE")){
   //Return HTTP 405 - Method Not Allowed
   crc.abortWith(Response.status(405).build());
   }
  }
}
```

Client side Response filters

Client side response filters are invoked after the HTTP response has been received from the server end but before it is dispatched to the caller/client. It provides an opportunity to mutate the properties of the HTTP response (Headers, Cookies etc) In order to implement a client side response filter, one needs to implement the extension interface provided by JAX-RS - *javax.ws.rs.client.ClientResponseFilter*

Client side Response filter

```
public class ClientResponseLoggerFilter {
 public void filter(ClientRequestContext reqCtx,
 ClientResponseContext resCtx) throws IOException{
  System.out.println("Response status: "+
  resCtx.getStatus());
 }
}
```

Sharing data between JAX-RS filters

JAX-RS API enables sharing of user-defined data amongst filters associated with a particular request

- It is abstracted in the form of a Map<String,Object> (pretty natural choice) via the *ContainerRequestContext*[20] interface
- Get all the custom properties using the *getPropertyNames()* method
- The value of a specific property can be fetched (from the *Map*) using *getProperty(String name)*
- Overwrite an existing property or a add a new one using *setProperty(String name, Object val)*

[20]https://docs.oracle.com/javaee/7/api/javax/ws/rs/container/ContainerRequestContext.html

The same capability is available in the *Client side* JAX-RS filters as well. The only difference is that you would be interacting with an instance of the *ClientRequestContext*[21]

Using custom (user defined) contextual data amongst JAX-RS request filters

```java
public class ReqFilter_1 implements ContainerRequestFilter {

  @Override
  public void filter(ContainerRequestContext cReqCtx) throws IOException {
    cReqCtx.setProperty("prop1", "value1");
  }
}

public class ReqFilter_2 implements ContainerRequestFilter {

  @Override
  public void filter(ContainerRequestContext cReqCtx) throws IOException {
    String val1 = (String) cReqCtx.getProperty("prop1");
    cReqCtx.setProperty("prop1", "value1");
    cReqCtx.setProperty("prop2", "value2");
  }
}

public class ReqFilter_3 implements ContainerRequestFilter {

  @Override
  public void filter(ContainerRequestContext cReqCtx) throws IOException {
    String val1 = (String) cReqCtx.getProperty("prop1");
    String val2 = (String) cReqCtx.getProperty("prop2");
    Collection<String> customProperties = cReqCtx.getPropertyNames();
  }
}
```

[21]https://docs.oracle.com/javaee/7/api/javax/ws/rs/client/ClientRequestContext.html

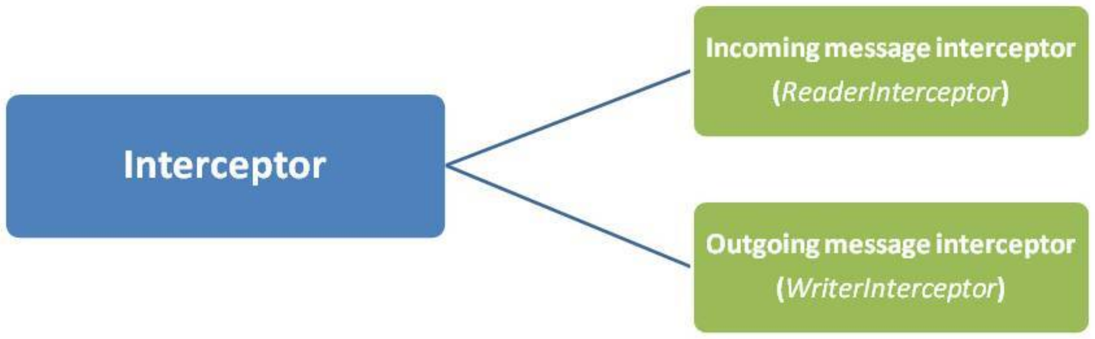
Sharing contextual data b/w Request and Response filters

```java
@Priority(Priorities.AUTHENTICATION)
public class ReqFilter_1 implements ContainerRequestFilter {

  @Override
  public void filter(ContainerRequestContext cReqCtx) throws IOException {
    //generated and used internally
    cReqCtx.setProperty("random-token", "token-007");
  }
}

public class ResponseFilter implements ContainerResponseFilter {

  @Override
  public void filter(ContainerRequestContext cReqCtx, ContainerResponseContext c\
RespCtx) throws IOException {
              //get the property
    String responseToken = (String) cReqCtx.getProperty("random-token");
    if(responseToken!=null){
      //set it to HTTP response
      cRespCtx.getHeaders.put("random-token-header" , responseToken); header
    }
  }
}
```

Interceptors

Interceptors are similar to filters in the sense that they are also used to mutate HTTP requests and responses, but the major difference lies in the fact that Interceptors are primarily used to manipulate HTTP message payloads. They are divided into two categories - *javax.ws.rs.ext.ReaderInterceptor* and *javax.ws.rs.ext.WriterInterceptor* for HTTP requests and responses respectively.

The same set of interceptors are applicable on the client side as well (unlike filters)


Interceptor
Incoming message interceptor (*ReaderInterceptor*)
Outgoing message interceptor (*WriterInterceptor*)

JAX-RS interceptors

Reader Interceptor

A ReaderInterceptor is a contract (extension interface) provided by the JAX-RS API. On the server side, a Reader Interceptor acts on HTTP payloads sent by the client while the client side reader interceptors are supposed to act on (read/mutate) the request payload prior to it being sent to the server

Writer Interceptor

On the server side, a Writer Interceptor act on HTTP payloads produced by the resource methods while the client side writer interceptors are supposed to act on (read/mutate) the payload sent by the server prior to it being dispatched to the caller

Interceptors are invoked in a chain like fashion (similar to filters). They are only triggered when entity providers (MessageBodyReader and MessageBodyWriter) are required to convert HTTP message to and from their Java object representations. Both ReaderInterceptor and WriterInteceptor wrap around MessageBodyReader and MessageBodyWriter respectively and hence executed in the same call stack.

Binding strategies for JAX-RS filters and interceptors

JAX-RS 2.0 defines multiple ways using which *server side* filters and interceptors can be bound to their target components.

- Global Binding
- Named Binding
- Dynamic Binding

Global Binding

By default, JAX-RS filters and interceptors are bound to *all* the methods of resource classes in an application. That is, both request (pre and post) and response filters will be invoked whenever any resource method is invoked in response to a HTTP request by the client. This convention can be overridden using named binding or dynamic binding.

Named Binding

Filters and interceptors scoping can be handled in a fine-grained manner (based on per resource class/method)

Configuring Named Binding

```
//Step 1: Define a custom annotation with the @NamedBinding annotation

@NameBinding
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(value = RetentionPolicy.RUNTIME)
public @interface Audited { }

//Step 2: Apply the custom annotation on the filter or interceptor

@Provider
@Audited
```

```
public class AuditFilter implements ContainerRequestFilter {
        //filter implementation....
}

//Step 3: Apply the same annotation to the required resource class or method

@GET
@Path("{id}")
@Produces("application/json")
@Audited
public Response find(@PathParam("id") String custId){
//search and return customer info
}
```

If it is applied to a class, the filter/interceptor will be bound to all its resource methods

Dynamic Binding

JAX-RS provides the *DynamicFeature* interface to help bind filters and interceptors dynamically at runtime. They can be used in tandem with the more static way of binding made possible using@NamedBinding. The injected instance of the *ResourceInfo* interface helps you choose the resource method in dynamic fashion by exposing various methods and the *FeatureContext* interface allows us to register the filter or interceptor once the resource method has been selected.

Dynamic Binding sample

```
@Provider
public class AuthFilterDynamic implements DynamicFeature {
    @Override
    public void configure(ResourceInfo resInfo, FeatureContext ctx) {
        if (UserResource.class.equals(resInfo.getResourceClass()) &&
           resInfo.getResourceMethod().getName().contains("PUT")) {
              ctx.register(AuthenticationFilter.class);
           }
      }
}
```

JAX-RS Providers: Part III

In this lesson, we are going to explore

- *Exception Mappers*: how to transform low application level exceptions to logical HTTP reponses
- *Context Provider*: provides context (of other classes) to resources and providers

Exception Mapper

The *javax.ws.rs.core.Response* object allows developers to wrap HTTP error state and return it to caller. A *javax.ws.rs.WebApplicationException* (unchecked exception) can be used as a bridge between the native business/domain specific exceptions and an equivalent HTTP error response. A *javax.ws.rs.ext.ExceptionMapper* represents a contract (interface) for a provider that maps Java exceptions to Response objects.

the benefits are obvious

- helps embrace "don't repeat yourself" (DRY) principle. your exception detection and transformation logic is not repeated across all of your service methods
- allows flexible mappings between business logic exceptions and the desired HTTP response
- in addition to returning HTTP status codes, you can choose to return HTTP payload in the message body as well

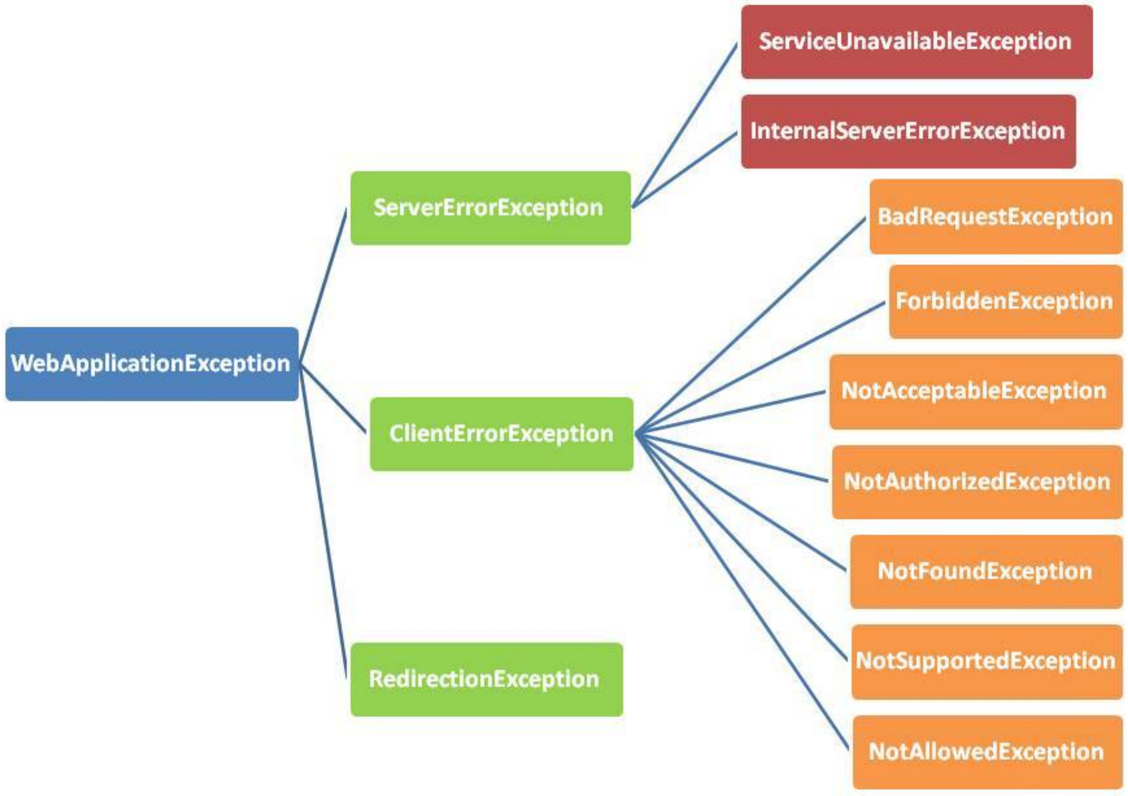
Custom exception mapper

```java
public class BookNotFoundMapper implements
  ExceptionMapper<BookNotFoundException>{
     @Override
      Response toResponse(
        BookNotFoundException bnfe){
           return Response.status(404).build();
      }
}
```

In the above example, thanks to the exception mapper, a *BookNotFoundException* is converted to a HTTP 404 response to the caller

New exception hierarchy

JAX-RS 2.0 has been supplemented with unchecked exceptions that inherit from javax.ws.rs.WebApplicationExceptio
This design relieves the developer from having to build and throw a WebApplicationException with explicit HTTP error information.



JAX-RS exception hierarchy

The new set of exceptions are intuitively named and each of them maps to a specific HTTP error scenario (by default) – this means that, throwing any of these exceptions from your resource classes would result in a pre-defined (as per mapping) HTTP error response being sent to the client e.g. the client would receive a HTTP 403 in case you throw a NotAuthorizedException. The exceptions' behavior can also be modified at runtime by using the ExceptionMapper to return a different Response object.

Exception	HTTP Error code
BadRequestException	400
ForbiddenException	403
InternalServerErrorException	500
NotAcceptableException	406
NotAllowedException	405
NotAuthorizedException	401
NotFoundException	404
NotSupportedException	415
ServiceUnavailableException	503

Context Provider

Context services are provided by the implementation of *ContextResolver* interface (which has just one method).

- Think of context as configuration metadata which can help with various aspects (instance creation etc.) and
- the Context Provider as a factory for these contexts

Here is a simple example

Simple Context Provider/Resolver

```
@Provider
@Produces("application/json")
public class MyCustomContextProvider implements ContextResolver<MyCustomContext>\
 {

        public MyCustomContext getContext(){
                return new MyCustomContext();
        }
}
```

Leveraging the Context Provider

```
public class Resource{

        @Context ctx
        Providers providers;

        @GET
        @Produces("application/json")
        public Response get(){

                //fetch it
                MyCustomContext customCtx = providers.
                getContextResolver(MyCustomContext.class
                        ,MediaType.APPLICATION_JSON);

                //use it
           MyCustomClass clazz = customCtx.create();
        }

}
```

JAX-RS for Power Users: Part I

In the upcoming chapters (including this one), we are going to bump things up a bit and go beyond the basics. This lesson covers

- JAX-RS *server side processing* pipeline
- *Request to method matching*

Server side request processing pipeline

The inspiration for this section was the **Processing Pipeline** section in the JAX-RS 2.0 specification doc[22] (Appendix C). I like it because of the fact that it provides a nice snapshot of all the modules in JAX-RS in the form of a ready to gulp capsule !

[22]http://download.oracle.com/otndocs/jcp/jaxrs-2_0_rev_A-mrel-eval-spec/index.html

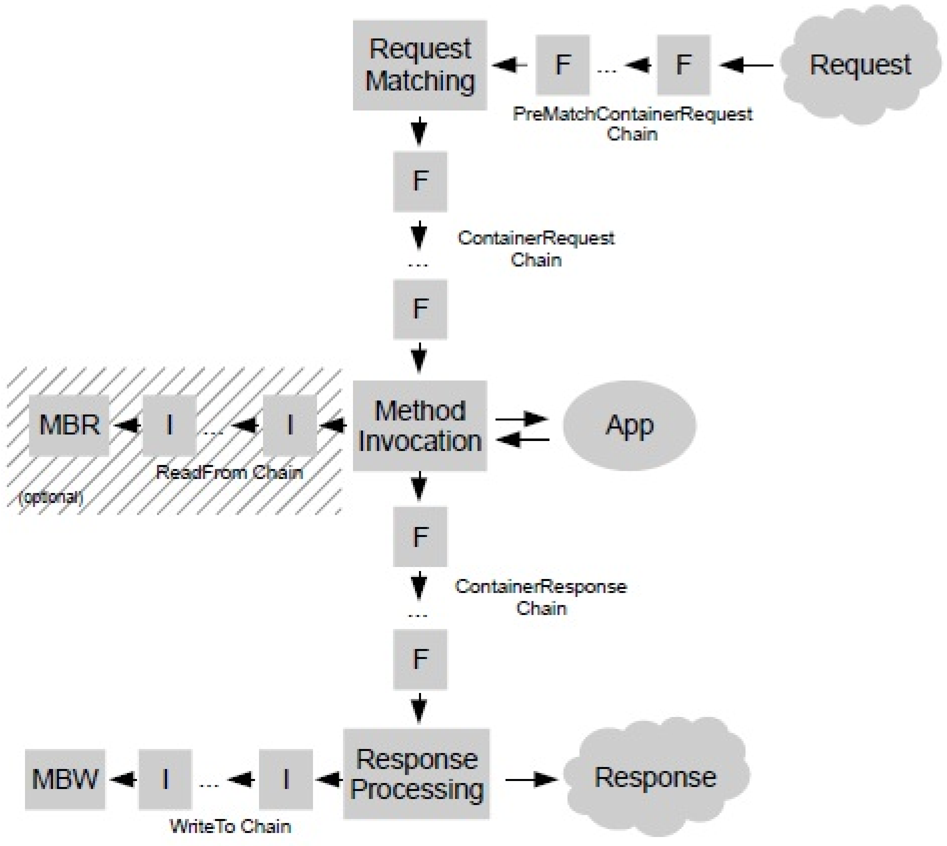


Figure C.1: JAX-RS Server Processing Pipeline.

The pipeline in all its glory

So I thought of using this diagram to provide a brief overview of the different JAX-RS components and how they orchestrate with each other. This chapter will make sense, now that we understand Entity Providers (Filters, Interceptors etc.)

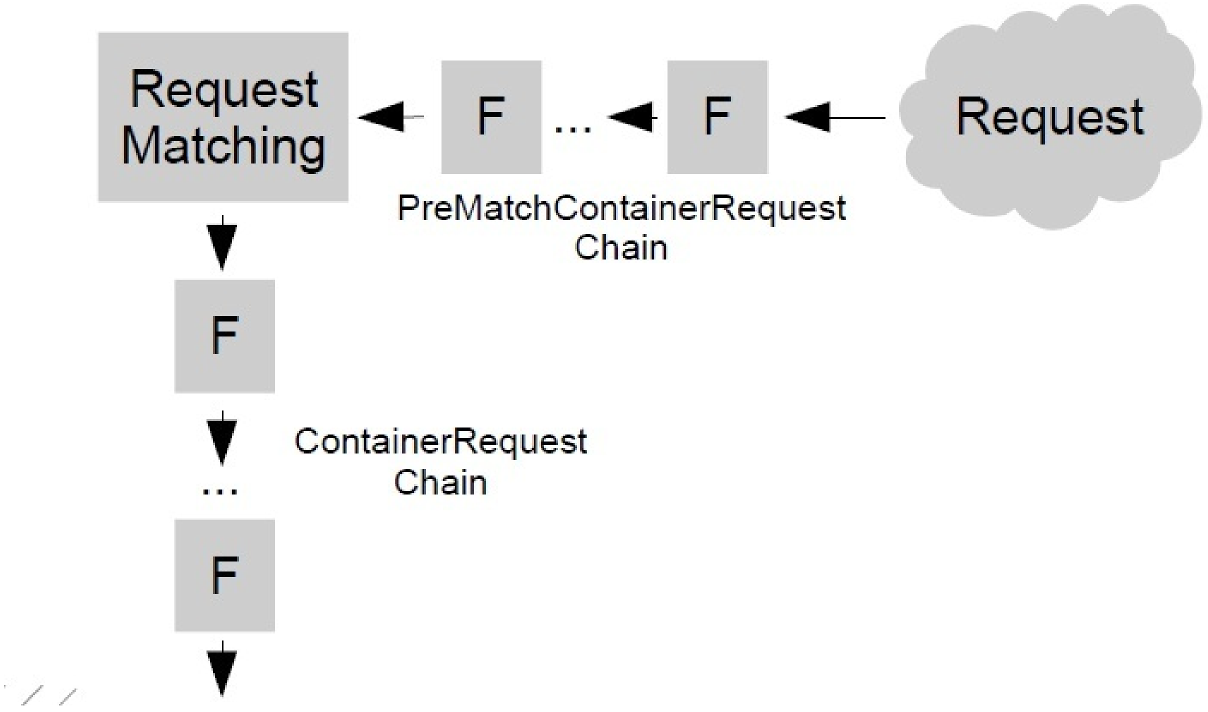
What's discussed here is the *server side* processing pipeline i.e. the sequence of actions which are triggered after the client sends an HTTP request (GET, POST, PUT etc). It all begins when the client (browser or custom REST client) sends an HTTP request to a REST endpoint

The goal is to educate you in terms of what happens behind the scenes and more importantly, in which order. The intricate details of components like filters, interceptors etc. have not been repeated since they were covered in the previous chapter(s)

Request chain

1. Request Filters

JAX-RS Filters are the first in a series of components which handle the HTTP request



Request filters

2. Method matching

After (successful) filter execution, the JAX-RS run time initiates the resource method matching process.

This chapter has a sub-section dedicated to this topic in case you want to explore it in details

Here is a quick peek

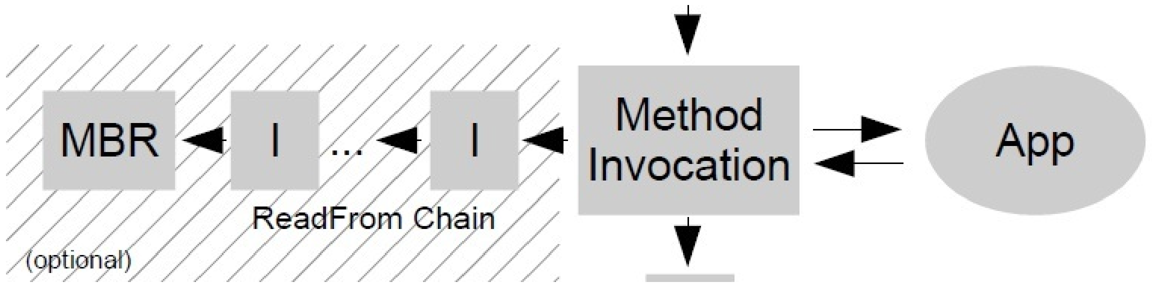
- The exact method to be invoked is based on the *algorithm* outlined by the specification (although JAX-RS providers are not bound by it)
- It's determined by a combination of below mentioned annotations
- @GET, @PUT, @POST, @DELETE etc - these are the annotations which should match up to the actual HTTP operation (the mapping of the annotation to the HTTP verb is rather obvious)

- @Path - its value (relative to the context root) is used to map the request URI e.g. /tweeters/all
- @Consumes - its values should match the *Content-Type* header value sent in the HTTP request
- @Produces - its values should match the *Accept* header value sent in the HTTP request

3. HTTP components injection

After the method matching is complete, the required HTTP components get injected into JAX-RS Resource classes (if configured) by the the JAX-RS run time. All we need to do is use the appropriate annotation

4. Request Interceptors



Request interceptors

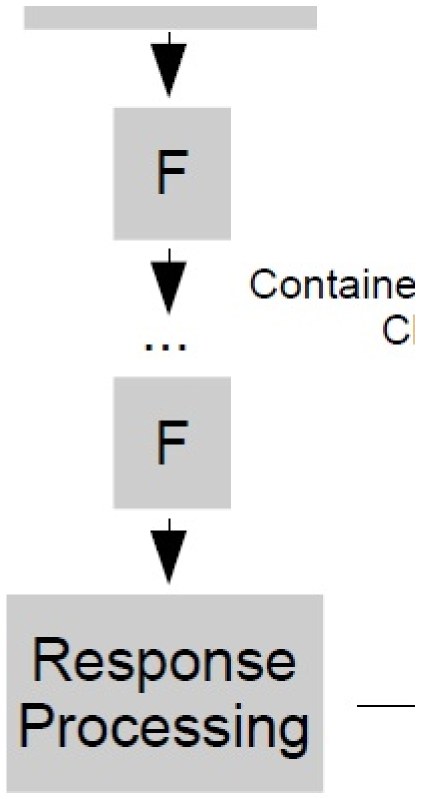
Interceptors come into play in case the HTTP payload is transformed by custom entity providers (Message Body Readers)

5. Entity Providers (*converting HTTP request payload to Java type*)

Entity Providers help in conversion of HTTP message payload to its appropriate Java type (for injection into the method parameters of JAX-RS resource classes) and vice versa

Response chain

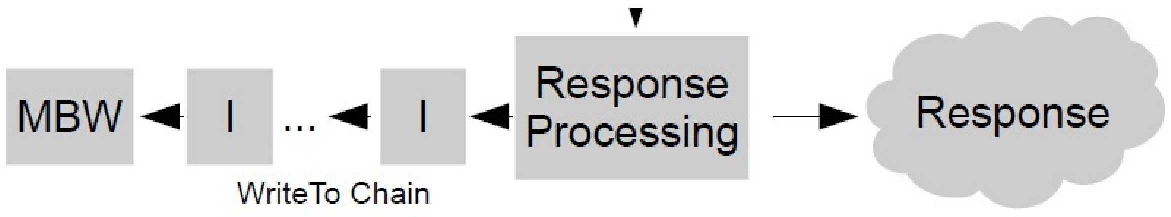
1. Response Filter



Response filters

Response Filters are similar to their Request-centric counterparts

2. Response Interceptors (chain)



response interceptors

They are invoked only when a *MessageBodyWriter* (see next topic) is registered to handle outgoing HTTP payload

3. Entity Providers

They deal with conversion of Java objects (within the application code) to HTTP response payloads

JAX-RS request to method matching

Let's look at the *HTTP request to resource method matching* in JAX-RS. Generally, the developers using the JAX-RS API are not exposed to (or do not really need to know) the nitty gritty of the *matching* process, rest assured that the JAX-RS runtime churns out its algorithms quietly in the background as our RESTful clients keep those HTTP requests coming! Just in case the term *request to resource method matching* is new to you - it's nothing but the process via which the JAX-RS provider dispatches a HTTP request to a particular method of your one of your resource classes (decorated with *@Path*)

Primary criteria

What are the factors taken into consideration during the request matching process ?

- HTTP request URI
- HTTP request method (GET, PUT, POST, DELETE etc)
- Media type of the HTTP request
- Media type of requested response

Here is what happens at runtime

- Narrow down the possible matching candidates to a set of resource classes. This is done by matching the HTTP request URI with the value of the *@Path* annotation on the resource classes
- From the set of resource classes in previous step, find a set of methods which are possible matching candidates (algorithm is applied to the filtered set of resource classes)
- The HTTP request verb is compared against the HTTP method specific annotations (@GET, @POST etc), the request media type specified by the *Content-Type* header is compared against the media type specified in the *@Consumes* annotation and the response media type specified by the *Accept* header is compared against the media type specified in the *@Produces* annotation.
- Boil down to the exact method which can serve the HTTP request

The above mentioned process is known as *Content Negotiation* and this topic is covered in details in another chapter

Further exploration

I would highly recommend looking at the Jersey server side logic[23] for implementation classes in the org.glassfish.jersey.server.internal.routing[24] package to get a deeper understanding. Some of the classes/implementation which you can look at are

- MatchResultInitializerRouter[25]
- SubResourceLocatorRouter[26]
- MethodSelectingRouter[27]
- PathMatchingRouter[28]

[23]https://github.com/jersey/jersey/tree/master/core-server

[24]https://github.com/jersey/jersey/tree/master/core-server/src/main/java/org/glassfish/jersey/server/internal/routing

[25]https://github.com/jersey/jersey/blob/master/core-server/src/main/java/org/glassfish/jersey/server/internal/routing/MatchResultInitializerRouter.java

[26]https://github.com/jersey/jersey/blob/master/core-server/src/main/java/org/glassfish/jersey/server/internal/routing/SubResourceLocatorRouter.java

[27]https://github.com/jersey/jersey/blob/master/core-server/src/main/java/org/glassfish/jersey/server/internal/routing/MethodSelectingRouter.java

[28]https://github.com/jersey/jersey/blob/master/core-server/src/main/java/org/glassfish/jersey/server/internal/routing/PathMatchingRouter.java

JAX-RS for Power Users: Part II

This chapter is about efficient JAX-RS. We'll explore

- *Content Negotiation*
- *Caching*: how does the JAX-RS framework help leverage the HTTP caching mechanism
- *Conditional access*: criteria based GET and PUT

HTTP Content Negotiation

Content Negotiation is the feature using which a client is able to *specify* certain characteristics of the response it is looking for.

> The term Content Negotiation is not JAX-RS specific. Rather, it is closely related to HTTP (hence, the web) and the JAX-RS framework supports the same

Must know

- *Criteria* (characteristics): the client can negotiate on the following properties of the HTTP response - encoding, media type, language
- Uses standard *HTTP headers*: Accept (media type), Accept-Language (language), Accept-Encoding (encoding)
- *Preferential selection* for Media types: uses additional metadata (*q*) to specify affinity for a particular media type from a list of multiple choices

Content Negotiation: the obvious way

Let's start off with an example

limited negotiation

```java
@Path("/books")
public class BooksResource {

    @Context
    HTTPHeaders headers;

    @GET
    @Produces("application/json", "application/xml")
    public Response all(){
      MediaType mType = headers.getAcceptableMediaTypes().get(0);
      Locale lang = headers.getAcceptableLanguages().get(0);
           Books books = getAll();

      return Response.ok(books).type(mType).language(lang).build();
    }
}
```

What's going on here?

- the JAX-RS resource offers JSON & XML media types (as per @Produces)
- the HTTP request headers are injected and the *getAcceptableMediaTypes* extracts the information from the *Accept* header
- *getAcceptableLanguages* does the same for language
- the extracted language and media type are set in the HTTP response as well

Drawback

- suitable for single negotiation values
- not fine grained in nature
- cannot handle complex/preferential negotiation

Content Negotiation: the fine grained way

Making use of the *Variant* API in JAX-RS can help. It provides a simple abstractions and can used with some of the advanced use cases where

- the client provides multiple choices e.g. more than one media type
- it specifies their weightage as well i.e. which one does it prefer more

fine grained negotiation

```
@Path("/books")
public class BooksResource {

    @Context
    Request req;

    @GET
    @Produces("application/json", "application/xml")
    public Response all(){
      List<Variant> variants = Arrays.asList(
      new Variant(MediaType.APPLICATION_XML_TYPE,"en", "deflate"),
      new Variant(MediaType.APPLICATION_JSON_TYPE,"en", "deflate")
      );

           //heavy lifting done by JAX-RS
           Variant theOne = req.selectVariant(variants);

           MediaType selectedMType = theOne.getMediaType();
           Locale lang = theOne.getLanguage();

      return Response.ok(books)
              .type(selectedMType).language(lang).build();
    }
}
```

Why is this better ??

- The above logic can handle complex preferential negotiation requests e.g. Accept: application/xml;q=1.0, application/json;q=0.5. This actually means, I prefer XML but JSON would work (in case you don't have XML data)
- The calculation is done by *Request#selectVariant* API - all you need to do is give it probable list from which to choose from

Implicit Content Negotiation based on Media Types

At its very core, JAX-RS drives media type based content negotiation on the basis of the Accept header sent by the client and the media type specified by @Produces annotation in the JAX-RS methods

Caching in JAX-RS

Caching is not a new concept. It is the act of storing data temporarily in a location from where it can be accessed faster (e.g. in-memory) as compared to it's original source (e.g. a database)

Before we dive in further, it's very important that we understand the following

- From a JAX-RS perspective, caching does not imply a server side cache
- It just provides *hints* to the client in terms of the durability/validity of the resource data
- It does not define how the client will use this hint. It ensures that it sticks to the HTTP semantics and assumes that the client (e.g. a browser, programmatic API based client etc.) understands the HTTP protocol

JAX-RS has had support for the Cache-Control header[29] was added in HTTP 1.1[30] since its initial (1.0) version. The CacheControl [31] class is an equivalent of the *Cache-Control* header in the HTTP world. It provides the ability to configure the header (and its different attributes) via simple setter methods.

So how to I use the *CacheControl* class?

Just return a Response [32]object around which you can *wrap* an instance of the CacheControl class.

Create a CacheControl instance manually and send it along with the Response

```java
@Path("/testcache")
public class RESTfulResource {
    @GET
    @Produces("text/plain")
    public Response find(){
      CacheControl cc = new CacheControl();
      cc.setMaxAge(20);
      return Response.ok(UUID.randomUUID().toString()).cacheControl(cc).build();
    }
}
```

At runtime

- The caller receives the Cache-Control response header and is free to use the information therein in order to decide when to fetch a new version of the resource

[29]http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html
[30]http://www.w3.org/Protocols/rfc2616/rfc2616.html
[31]http://docs.oracle.com/javaee/7/api/javax/ws/rs/core/CacheControl.html
[32]http://docs.oracle.com/javaee/7/api/javax/ws/rs/core/Response.html

- If the *max-age* attribute has a value of 3600 seconds, it means that the client is free to cache the resource representation for the next one minute (and does not need to call upon the server)

Please note that the JAX-RS framework does not define how the client will actually 'cache' the response

CDI Producers

We can use CDI to enforce caching semantics in a declarative manner. CDI [33]Producers can help inject instances of classes which are *not technically beans* (as per the strict definition) or for classes over which you do not have control as far as decorating them with scopes and qualifiers are concerned. The idea is to

- Have a custom annotation (*@CacheControlConfig*) to define default values for Cache-Control header and allow for flexibility in case you want to override it
- Use a CDI Producer to create an instance of the CacheControl class by using the *InjectionPoint* object (injected with pleasure by CDI !) depending upon the annotation parameters
- Just *inject* the CacheControl instance in your REST resource class and use it in your methods

A custom annotation to configure Cache Control parameters

```
@Retention(RUNTIME)
@Target({FIELD, PARAMETER})
public @interface CachControlConfig {

    public boolean isPrivate() default true;
    public boolean noCache() default false;
    public boolean noStore() default false;
    public boolean noTransform() default true;
    public boolean mustRevalidate() default true;
    public boolean proxyRevalidate() default false;
    public int maxAge() default 0;
    public int sMaxAge() default 0;

}
```

[33] cdispec.org

A CDI Producer (factory)

```java
public class CacheControlFactory {

    @Produces
    public CacheControl get(InjectionPoint ip) {

        CachControlConfig ccConfig = ip.getAnnotated()
        .getAnnotation(CachControlConfig.class);

        CacheControl cc = null;
        if (ccConfig != null) {
            cc = new CacheControl();
            cc.setMaxAge(ccConfig.maxAge());
            cc.setMustRevalidate(ccConfig.mustRevalidate());
            cc.setNoCache(ccConfig.noCache());
            cc.setNoStore(ccConfig.noStore());
            cc.setNoTransform(ccConfig.noTransform());
            cc.setPrivate(ccConfig.isPrivate());
            cc.setProxyRevalidate(ccConfig.proxyRevalidate());
            cc.setSMaxAge(ccConfig.sMaxAge());
        }

        return cc;
    }
}
```

Good to go!

```java
@Path("/testcache")
public class RESTfulResource {
    @Inject
    @CachControlConfig(maxAge = 20)
    CacheControl cc;

    @GET
    @Produces("text/plain")
    public Response find() {
        return Response.ok(UUID.randomUUID()
        .toString()).cacheControl(cc).build();
    }
}
```

Additional thoughts

- In this case, the scope of the produced CacheControl instance is *@Dependent* i.e. it will live and die with the class which has injected it. In this case, the JAX-RS resource itself is *RequestScoped* (by default) since the JAX-RS container creates a new instance for each client request, hence a new instance of the injected CacheControl instance will be created along with each HTTP request
- You can also introduce *CDI qualifiers* to further narrow the scopes and account for corner cases
- You might think that the same can be achieved using a JAX-RS filter. That is correct. But you would need to set the Cache-Control header manually (within a mutable MultivaluedMap) and the logic will not be flexible enough to account for different Cache-Control configurations for different scenarios

Efficient JAX-RS: Conditional GETs & PUTs

This section discusses how to leverage features in the JAX-RS API to execute RESTful operations based on conditions/criteria in order to aid with *scalability* and *performance*. It covers

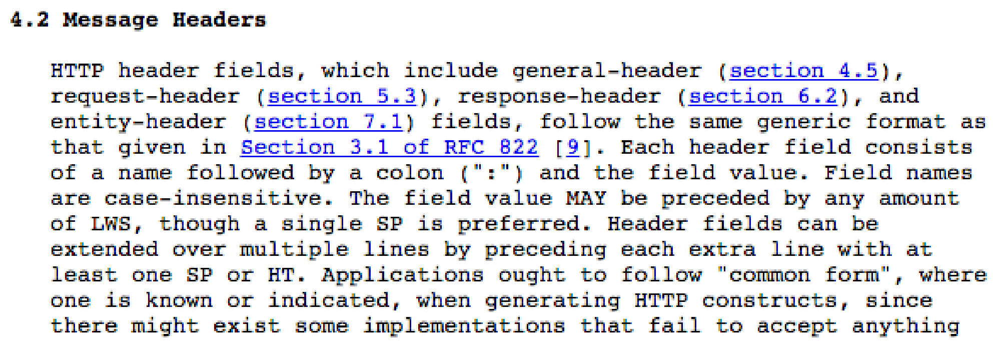
- which HTTP headers are involved
- which JAX-RS APIs to use
- details of the entire request-response flow

Must know HTTP headers

In addition to the Cache-Control HTTP header, we would also encounter few others in this section. If you haven’t heard of these before, don’t worry, their names are self-explanatory :-)

- Last-Modified
- If-Modified-Since
- If-Unmodified-Since
- ETag
- If-None-Match

It would be awesome if you have a basic understanding of (at least some of) these headers. These are best referred from the official HTTP specification document[34].



```
4.2 Message Headers

   HTTP header fields, which include general-header (section 4.5),
   request-header (section 5.3), response-header (section 6.2), and
   entity-header (section 7.1) fields, follow the same generic format as
   that given in Section 3.1 of RFC 822 [9]. Each header field consists
   of a name followed by a colon (":") and the field value. Field names
   are case-insensitive. The field value MAY be preceded by any amount
   of LWS, though a single SP is preferred. Header fields can be
   extended over multiple lines by preceding each extra line with at
   least one SP or HT. Applications ought to follow "common form", where
   one is known or indicated, when generating HTTP constructs, since
   there might exist some implementations that fail to accept anything
```

Before we proceed…

Here is a quickie on the two important JAX-RS APIs which we will be discussing here

- EntityTag[35]: JAX-RS equivalent (simple class) of the HTTP ETag header
- Request[36]: the main API which contains utility methods to evaluate the conditions which in turn determine the criteria for access

Cache Revalidation: that's what it's all about

A simple interaction with a JAX-RS service can be as follows

- Client sends a GET request
- Server replies back with the requested resource (with a HTTP 200 status)
- It also sends the *Cache-Control* & *Last-Modified* headers in response

Cache-Control defines the *expiration semantics* (along with other fine grained details) for the resource on the basis of which the client would want to

- *revalidate* it's cache i.e. invoke the GET operation for same resource (again)
- make sure it does so in an *efficient/scalable/economic* manner i.e. not repeat the same process of exchanging data (resource info) if there are no changes to the information that has been requested

Common sense stuff right ? Let's look at how we can achieve this

Leverage the *Last-Modified* and *If-Modified-Since* headers

[34]https://tools.ietf.org/html/rfc2616#section-4.2
[35]http://docs.oracle.com/javaee/7/api/javax/ws/rs/core/EntityTag.html
[36]http://docs.oracle.com/javaee/7/api/javax/ws/rs/core/Request.html

Improving GET request performance in JAX-RS by using the Last-Modified and If-Modified-Since headers

```java
@Path("books")
@Stateless
public class BooksResource_1{

  @PersistenceContext
  EntityManager em;

  @Context
  Request request;

  @Path("{id}")
  @GET
  @Produces("application/json")
  public Response getById(@PathParam("id") String id){
    //get book info from backend DB
    Book book = em.find(Book.class, id);
    //get last modified date
    Date lastModified = book.getLastModified();

    //let JAX-RS do the math!
    ResponseBuilder evaluationResultBuilder = request.evaluatePreconditions(last\
Modified);

    if(evaluationResultBuilder == null){
      //resource was modified, send latest info (and HTTP 200 status)
      evaluationResultBuilder = Response.ok(book);
    }else{
      System.out.println("Resource not modified - HTTP 304 status");
    }
    CacheControl caching = ...; //decide caching semantics
    //add metadata
    evaluationResultBuilder.cacheControl(caching)
                           .header("Last-Modified",lastModified);

    return evaluationResultBuilder.build();
  }
}
```

- Server sends the Cache-Control & Last-Modified headers as a response (for a GET request)

- In an attempt to refresh/revalidate it's cache, the client sends the value of the Last-Modified header in the *If-Modified-Since* header when requesting for the resource in a subsequent request
- *Request#evaluatePreconditions(Date)*[37] determines whether or not the value passed in the If-Modified-Since header is the same as the date passed to the method (ideally the modified date would need to extracted from somewhere and passed on this method)

ETag in action

Improving GET request performance in JAX-RS by using the ETag header

```
@Path("books")
@Stateless
public class BooksResource_2{

  @PersistenceContext
  EntityManager em;

  @Context
  Request request;

  @Path("{id}")
  @GET
  @Produces("application/json")
  public Response getById(@PathParam("id") String id){
    Book book = em.find(Book.class, id); //get book info from backend DB
    //calculate tag value based on your custom implementation
    String uniqueHashForBook = uniqueHashForBook(book);
    //instantiate the object
    EntityTag etag = new EntityTag(uniqueHashForBook)

    //let JAX-RS do the math!
    ResponseBuilder evaluationResultBuilder = request.evaluatePreconditions(etag\
);

    if(evaluationResultBuilder == null){
      //resource was modified, send latest info (and HTTP 200 status)
      evaluationResultBuilder = Response.ok(book);
    }else{
      System.out.println("Resource not modified - HTTP 304 status");
    }
```

[37] http://docs.oracle.com/javaee/7/api/javax/ws/rs/core/Request.html#evaluatePreconditions-java.util.Date-

```
    CacheControl caching = ...; //decide caching semantics
    evaluationResultBuilder.cacheControl(caching)
                           .tag(etag); //add metadata

    return evaluationResultBuilder.build();
  }
}
```

- In addition to the Last-Modified header, the server can also set the ***ETag*** header value to a string which uniquely identifies the resource and changes when it changes e.g. a hash/digest
- client sends the value of the ETag header in the ***If-None-Match*** header when requesting for the resource in a subsequent request
- and then its over to the *Request#evaluatePreconditions(EntityTag)*[38]

With the Request#evaluatePreconditions(Date,EntityTag) [39]the client can use *both* last modified date as well as the ETag values for criteria determination. This would require the client to set the If-Modified-Since header*

Making use of the API response…

In both the scenarios

- if the *Request#evaluatePreconditions* method returns null, this means that the *pre-conditions were met* (the resource was modified since a specific time stamp and/or the entity tag representing the resource does not match the specific ETag header) and the *latest version of the resource must be fetched* and sent back to the client
- otherwise, a *HTTP 304 (Not Modified)* response is automatically returned by the method, and it can be returned as is

Please note …

- *Choice of ETag*: this needs to be done carefully and depends on the dynamics of your application. *What are the attributes of your resource whose changes are critical for your clients ?* Those are the ones which you should use within your ETag implementation
- *Not a magic bullet*: based on the precondition evaluation, you can help prevent unnecessary exchange of data b/w client and your REST service layer, but not between your JAX-RS service and the backend repository (e.g. a database). It's important to understand this

[38]http://docs.oracle.com/javaee/7/api/javax/ws/rs/core/Request.html#evaluatePreconditions-javax.ws.rs.core.EntityTag-
[39]http://docs.oracle.com/javaee/7/api/javax/ws/rs/core/Request.html#evaluatePreconditions-java.util.Date-javax.ws.rs.core.EntityTag-

Can I only improve my GETs …?

No ! the HTTP spec cares abut *PUT* operations as well; and so does the JAX-RS spec :-)

Improving PUTs

```
@Path("books")
@Stateless
public class BooksResource_3{

  @PersistenceContext
  EntityManager em;

  @Context
  Request request;

  @Path("{id}")
  @PUT
  public Response update(@PathParam("id") String id, Book updatedBook){
    Book book = em.find(Book.class, id); //get book info from backend DB
    Date lastModified = book.getLastModified(); //get last modified date
    //let JAX-RS do the math!
    ResponseBuilder evaluationResultBuilder = request.evaluatePreconditions(last\
Modified);

    if(evaluationResultBuilder == null){
      em.merge(updatedBook); //no changes to book data. safe to update book info
      //(ideally) nothing needs to sent back to the client in case of successful
      evaluationResultBuilder = Response.noContent(); update
    }else{
      System.out.println("Resource was modified after specified time stamp - HTT\
P 412 status");
    }

    CacheControl caching = ...; //decide caching semantics
    evaluationResultBuilder.cacheControl(caching)
                           .header("Last-Modified",lastModified); //add metadata

    return evaluationResultBuilder.build();
}
```

- Server sends the Cache-Control & Last-Modified headers as a response (for a GET request)

- In an attempt to send an updated value of the resource, the client sends the value of the Last-Modified header in the *If-Unmodified-Since* header
- *Request#evaluatePreconditions(Date)* method determines whether or not the value passed in the If-Unmodified-Since header is the same as the date passed to the method (in your implementation)

here is the gist …

- If the API returns a non null response, this means that the pre-conditions were not met (HTTP 412) i.e. the resource was in fact modified after the time stamp sent in the If-Unmodified-Since header, which of course means that the caller has a (potentially) *stale* (outdated) version of the resource
- Otherwise (for a null output from the API), its a hint for the client to go ahead and execute the update operation
- In this scenario, what you end up saving is the cost of update operation executed against your database in case the client's version of the resource is outdated

Asynchronous JAX-RS

This chapter covers asynchronous programming support in JAX-RS and some of its potential gotchas

Basics of server side async JAX-RS

JAX-RS 2.0 includes a brand new API for *asynchronous processing* which includes server as well as client side counterparts. Being asynchronous inherently implies request processing on a *different thread* than that of the thread which initiated the request

- From a *client* perspective, it prevents blocking the request thread since no time is spent waiting for a response from the server.
- Similarly, asynchronous processing on the *server* side involves suspension of the original request thread and initiation of request processing on a different thread, thereby freeing up the original server side thread to accept other incoming requests.

The end result of asynchronous execution (if leveraged correctly) is scalability, responsiveness and greater throughput.

Server side async

On the server side, asynchronous behaviour is driven by

- *@Suspended*: annotation which instructs the container to inject an instance of AsyncResponse and invoke the method asynchronously
- *AsyncResponse*: bridge between the application logic and the client request

An instance of AsyncResponse can be transparently injected as a method parameter (of a JAX-RS resource class) by annotating it with @Suspended. This instance serves as a callback object to interact with the caller/client and perform operations such as response delivery (post request processing completion), request cancellation, error propagation, and so forth

Async JAX-RS in action

```
@Path("async")
@Stateless
public class AsyncResource {

 @Resource
 ManagedExecutorService mes;

 @GET
  public void async(@Suspended AsyncResponse ar) {

     String initialThread = Thread.currentThread().getName();
     System.out.println("Thread: "+ initialThread + " in action...");

        mes.execute(new Runnable() {
         @Override
         public void run() {
          try {
             String processingThread = Thread.currentThread().getName();
             System.out.println("Processing thread: " + processingThread);

             Thread.sleep(5000);
             String respBody = "Process initated in "
             + initialThread + " and finished in " + processingThread;

             ar.resume(Response.ok(respBody).build());                        \

              }
              catch (InterruptedException ex) {
              //ignored. . . don't try this in production!
                  }
                }
            });

        System.out.println(initialThread + " freed ...");
    }
}
```

Since we have clearly expressed our asynchronous requirements, the container will ensure that

- the calling thread is *released*

- the actual business logic is executed in a *different* thread (in this case its the thread pool taken care of the by the Managed Executor Service[40] in the Java EE container - thanks to Concurrency Utilties[41] in Java EE 7)

Things to watch out for

Although the calling (request) thread is released, *the underlying I/O thread still blocks until the processing in background thread continues.*In the above example, the caller would have to wait 5 seconds since that's the delay we have purposefully introduced within the code. In simple words, the client (e.g. browser executing a HTTP GET) keeps waiting until the business logic execution is finished by calling the ***resume*** method of the injected *AsyncResponse* object

Timeouts to the rescue

One can specify a time out period after which the client gets back a *HTTP 503 Service Unavailable* response (*default* convention)

Basic async timeout config

```java
@Path("async")
@Stateless
public class AsyncResource {

    @Resource
    ManagedExecutorService mes;

    @GET
    public void async(@Suspended AsyncResponse ar) {
        ar.setTimeout(3, TimeUnit.SECONDS); //setting the time out to 3 seconds
        String initialThread = Thread.currentThread().getName();
        ......
    }
}
```

Fine grained time outs

The default behaviour (HTTP 503 on time out) might not be suitable for all use cases. For example, you might want to implement a solution where a tracking identifier needs to be sent to the client (for future) if the actual processing does not finish in due time (before timeout triggers). Having

[40]https://docs.oracle.com/javaee/7/api/javax/enterprise/concurrent/ManagedExecutorService.html
[41]https://jcp.org/ja/jsr/detail?id=236

the ability to send a custom HTTP response on time out can prove useful. The AsyncResponse API makes this possible via the notion of a *time out handler*[42]. You can do this by

- a direct HTTP response (see below example)
- via an exception (by passing an instance of *Throwable* to *AsyncResponse#resume*[43])

fine grained timeout config

```
@GET
public void async(@Suspended AsyncResponse ar) {
  ar.setTimeout(3, TimeUnit.SECONDS);
  ar.setTimeoutHandler(new TimeoutHandler() {
    @Override
      public void handleTimeout(AsyncResponse asyncResponse) {
          //sending HTTP 202 (Accepted)
          asyncResponse.resume(Response
          .accepted(UUID.randomUUID().toString())
          .build());
      }
});
```

Client side async using the JAX-RS Client API

Using asynchronous behaviour on the client side is pretty easy. All you need to do is obtain an instance of AsyncInvoker by calling ***async***[44] on the *Invocation.Builder*

Client side async

```
Client client = ClientBuilder.newBuilder().build();
WebTarget target = client.target("https://api.github.com/search/users?q=abhirock\
zz");

Invocation.Builder reqBuilder = target.request();
AsyncInvoker asyncInvoker = reqBuilder.async();
Future<Response> futureResp = asyncInvoker.get();

Response response = futureResp.get(); //blocks until client responds or times out
String responseBody = response.readEntity(String.class);
```

[42]http://docs.oracle.com/javaee/7/api/javax/ws/rs/container/TimeoutHandler.html
[43]http://docs.oracle.com/javaee/7/api/javax/ws/rs/container/AsyncResponse.html#resume-java.lang.Throwable-
[44]http://docs.oracle.com/javaee/7/api/javax/ws/rs/client/Invocation.Builder.html#async--

AsyncInvoker interface supports asynchronous invocation with dedicated methods (get(), post(), put() and so on) for standard HTTP actions: GET, PUT, POST, DELETE, HEAD, TRACE, and OPTIONS.

Response handling via Callbacks and Futures

Once registered, an implementation of the *InvocationCallback* will automatically be executed once the asynchronous request is processed. It provides the ability to account for both successful and exceptional scenarios. In the event a Future object is obtained and no callback has been registered, manually poll it in order to interact with the response. *isDone*, *get*, *cancel* are some of the methods that can be invoked

Client side async callback

```
Client client = ClientBuilder.newBuilder().build();
WebTarget target = client.target("https://api.github.com/search/users?q=abhirock\
zz");

Invocation.Builder reqBuilder = target.request();
Invocation invocation = reqBuilder.buildGet();

Future<Response> future2 =
  invocation.submit(
    new InvocationCallback<Customer>(){
        public void completed(Customer cust){
        System.out.println(
          "Customer ID:" + cust.getID());
        }
         public void failed(Throwable t){
        System.out.println(
          "Unable to fetch Cust details: " +
          t.getMessage());
      }
});
```

Client side gotchas

As you might have already observed, async behaviour in (server side) JAX-RS is not the same as in other typical async APIs i.e. the client is still blocked. One should use the *async* method (as demonstrated above) to call a server side REST API in an async manner even if the server REST API itself is asynchronous in nature (implemented using *@Suspended AsyncResponse*) .

Do not remain under the impression that your client thread will return immediately just because the server API is async

Client side async gotcha!

```java
public Response test() throws Exception{
    Client client = ClientBuilder.newBuilder().build();
    WebTarget target = client.target("http://localhost:8080/jaxrs-async-service/\
async");

    Invocation.Builder reqBuilder = target.request();
    //this will block until server responds or triggers out (even if its aysnc)
    Response response = reqBuilder.get();

    String responseBody = response.readEntity(String.class);
    return Response.status(response.getStatus()).entity(responseBody).build();
}
```

The call to async returns an instance of *Future*[45] object - and you might already know, the *get*[46] method (of Future) *blocks*. So use it with care and at the correct point in your application logic

[45]http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html?is-external=true

[46]http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html#get()

JAX-RS.next()

JAX-RS is a key specification of the Java EE Platform. Some of the potential additions to the next version of JAX-RS 2.1[47] slated for Java EE 8[48] are as follows

What's discussed in this lesson is subject to change

Integration based

There are a lot of additions being planned which involve integration with other APIs (including some of the new ones in Java EE 8)

Joining hands with JSON-B

Currently, JAX-RS provides seamless, out-of-the-box support for JAXB as well as JSON-P objects. The JSON-B[49] specification will empower with support for runtime conversion between Java model and JSON data based on statically defined (development time) bindings. JAX-RS will provide this as an inbuilt/integrated feature. All you would need to is define your model classes with JSON-B annotations and leave the rest to the JAX-RS run time!

JAX-RS & MVC 1.0

MVC 1.0[50] is another Java EE 8 candidate. JAX-RS is important from an integration perspective, since JAX-RS resource classes are designed to play the role of *Controller*

Explicit support for Security

As you might already read in the previous chapter, JAX-RS relies on the security imposed and implement by the Java EE Web Container (Servlet specification). But the goal for Java EE 8 is to either standardize JAX-RS specific security features or leverage Java EE Security 1.0[51] (yet another Java EE 8 spec)

[47] https://jcp.org/en/jsr/detail?id=370
[48] https://jcp.org/en/jsr/detail?id=366
[49] https://www.jcp.org/en/jsr/detail?id=367
[50] https://www.jcp.org/en/jsr/detail?id=371
[51] https://www.jcp.org/en/jsr/detail?id=375

Tighter integration with CDI

CDI 2.0[52] is in the works for Java EE 8. A tighter integration with CDI implies usage of *@Inject* in a more widespread manner (as opposed to JAX-RS specific injection constructs such as @Context etc.). This is more from a platform unification perspective

Support for Server Sent Events (SSE)

Inclusion of a standard API for Server Sent Events was well what the community had asked for as evident in page 2 of the Java EE 8 Community survey results[53]. If you havenâ€™t heard of or know about SSE, think of it as a middle path b/w HTTP (request-response based) and WebSockets (full duplexed and bi-directional).

Jersey provides support for SSE but it is not a part of the JAX-RS standard and hence not portable. You can read more on this here[54]

Enabling NIO (non blocking I/O) for JAX-RS providers

This is to complement the already existing Async (server & client) capabilities available since JAX-RS 2.0

Other features

- support for *reactive* programming via JAX-RS
- improving *HATEOAS* support

[52]https://jcp.org/en/jsr/detail?id=365
[53]https://java.net/downloads/javaee-spec/JavaEE8_Community_Survey_Results.pdf
[54]https://jersey.java.net/documentation/latest/sse.html