The JAX-RS API forms an important part of the Java EE platform's commitment to providing standards-driven technology. The ubiquitous nature of the Internet and that recent increasing interest in the microservice architecture has put more focus on small, scalable, autonomous services and their interoperability. The principal methodology used to allow microservices to communicate with each other and the 'outside world' is REST and its use in developing RESTful APIs. The technology that Java EE provides for this is the **JAX-RS: Java API for RESTful Web Services**.

The Goals of JAX-RS

The goals of the JAX-RS API are:

- **POJO-based:** To provide a collection of classes/interfaces and associated annotations to be used with POJOs so as to expose them as Web resources.
- **HTTP-centric:** To use HTTP as the underlying network protocol and provide a clear mapping between HTTP and URI elements and the corresponding API classes and annotations.
- **Format independence:** To be applicable to a wide variety of HTTP entity body content types and provide the necessary pluggability to allow additional types to be added.
- **Container independence:** To ensure that artifacts using the API are deployable in a range of Web servers.
- **Inclusion in Java EE:** To allow Java EE features and components to be used within a Web resource class.

Overview of JAX-RS Annotations

Annotations in the JAX-RS API are used to provide meta-data around the web resource. A typical example is to use the @GET annotation with the @Path annotation to identify the method that should handle a GET request to the specified URI in the @Path annotation.

What follows is a very quick overview of the annotations available to mark the methods and classes used to construct web resources. This is not an exhaustive list, there are a few more annotations in the JAR-RS arsenal, however, as the majority of the work of JAX-RS is in configuration and handling web resources, this is where you will find the majority of the API's annotations put to use.

This is the first in a three-part series looking at JAX-RS annotations.

Part two covers:

- The @Path Annotation and @PathParam
- The @QueryParamter Annotation
- The @Produces Annotation
- The @Consumes Annotation

Part three covers:

- The @FormParam Annotation
- The @MatrixParam Annotation
- The @CookieParam Annotation

- The @HeaderParam Annotation
- The @Provider Annotation

The @Path Annotation

The annotation identifies the URI path template to which the resource responds, and this annotation is specified at the class level of a resource. The annotation's value is a partial URI path template relative to the base URI of the server on which the resource is deployed, the context root of the WAR, and the URL pattern to which the server helper servlet responds.

Now let us see this annotation with an example

```
@Path("/employees/{employeename}")

public class EmployeeResource {

   @GET

   @Produces("text/xml")

   public String getEmployee(@PathParam("employeename") String employeeName)
{

      ...

   }

}
```

If it is required that a employee name must only consist of lower and upper case numeric characters, it is possible to declare a particular regular expression that will override the default regular expression, "[^/]+?". The following example shows how this could be used with the annotation.

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

The @GET HTTP Method Annotation

The **GET** method means retrieve whatever information (in the form of an entity) is identified by the Request-URI. If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public Response getAllBooks() {
   List<Book> books = BookRepository.getAllBooks(); // queries database for all books
   GenericEntity<List<Book>> list = new GenericEntity<List<Book>>(books) {};
   return Response.ok(list).build();
}
```

What is the difference between these two types of spring GET methods?

Only one of the two examples you posted is an actual *Spring* GET, and that is the second one @RequestMapping(value = "/hello", method = RequestMethod.GET) annotation. It is a Spring MVC implementation.

The other one, the first @GET @Path("/hello"), is actually a *JAX-RS* GET specification, and you will need an implementation of the JAX-RS to make it work.
The main differences between them are not on the "GET" only, but on the overall frameworks. There are already detailed articles like this one covering the differences between JAX-RS and Spring MVC RESTful.

Since REST is not a formal specification, implementations of it may vary slightly from one provider from another, but the concept is the same.

Which one is the preferred method?

Spring MVC's RESTful will be more tightly integrated into the Spring Framework.

JAX-RS will follow the Java EE implementation, and the integration will benefit a full Java EE environment.

And you can also "merge" Spring and Java EE as well. There are connectors that can integrate Spring and JAX-RS so you can benefit from one or another. See this example.
So the preferred method depends. Generally speaking:

- If you are running in a full Java EE container (Like JBoss or Glassfish), use **JAX-RS** method. The implementation for JAX-RS will be already avaible in the environment.
- If you are running in a Spring IoC and wanna stay detached from Java EE use **Spring MVC RESTful** method.
- If you are using a Java EE Web Profile, like Tomcat, you gonna have to decide based on specific criteria, like which interface framework you will be using, and for that purpose you are building the application.

I found Spring easier to configure and set up in Tomcat, but that is my opinion. Also the Spring exception handling @ControllerAdvice and @ExceptionHandler solved my RESTful JSON handling perfectly for what I was needing, but maybe there is something similiar for JAX-RS too.
As a final statement I think you should define your RESTful framework (JAX-RS, Spring, or even another one) based mainly on which environment you are going to

run on, but also considering all the integrations and resources you are going to need for yor project.

The @POST HTTP Method Annotation

Methods annotated @POST respond to POST method requests.

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response saveBook(Book book) {
   book = bookRepository.saveBook(book);
   return Response.ok(book).build();
}
```

The POST HTTP method is commonly used to create a resource. This example code persists the new book object in the database.

The @PUT HTTP Method Annotation

The @PUT annotation is used for updating a record and method annotated this way respond to an HTTP PUT request.

```
@PUT
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response updateBook(Book book) {
   book = bookRepository.updateBook(book);
   return Response.ok(book).build();
}
```

The @DELETE HTTP Method Annotation

Methods annotated @DELETE are expected to delete a resource.

```
@DELETE
@Path("{isbn}")
@Produces(MediaType.APPLICATION_JSON)
public Response deleteBook(@PathParam("isbn") String isbn) {
   Book book = bookRepository.deleteBookByIsbn(isbn);
   return Response.ok(book).build();
}
```

Usually, the resource or its id is passed to the resource method parameter from the URI variable as you can see in this example.

The @OPTIONS HTTP Method Annotation

Methods annotated with @OPTIONS respond to HTTP Option requests.

```
@OPTIONS
```

```
public Response preflight() {
   return Response.ok().header("Allow", true).build();
}
```

The options method is used as a request when the client wishes to make a complex HTTP request to a different domain. It is done in order to determine if the client is allowed to make the request or not.

The @HEAD HTTP Method Annotation

The HTTP HEAD method is identical to HTTP GET method except that the server mustn't respond with a body in the response.

```
@HEAD
public Response headsUp() {
   return Response.ok().build();
}
```

This method is to obtain meta-data regarding the entity without sending back the entity body itself.

Spring @RequestMapping

1. **@RequestMapping with Class**: We can use it with class definition to create the base URI. For example:

```
@Controller
@RequestMapping("/home")
public class HomeController {

}
```

   Now /home is the URI for which this controller will be used. This concept is very similar to servlet context of a web application.

7. **@RequestMapping with Method**: We can use it with method to provide the URI pattern for which handler method will be used. For example:

```
@RequestMapping(value="/method0")
@ResponseBody
public String method0(){
    return "method0";
}
```

   Above annotation can also be written as @RequestMapping("/method0"). On a side note, I am using @ResponseBody to send the String response for this web request, this is done to keep the example simple. Like I always do, I will use these methods in Spring MVC application and test them with a simple program or script.

13. **@RequestMapping with Multiple URI**: We can use a single method for handling multiple URIs, for example:

```
@RequestMapping(value={"/method1","/method1/second"})
@ResponseBody
public String method1(){
```

```
    return "method1";
}
```

If you will look at the source code of **RequestMapping annotation**, you will see that all of it's variables are arrays. We can create String array for the URI mappings for the handler method.

19. **@RequestMapping with HTTP Method**: Sometimes we want to perform different operations based on the HTTP method used, even though request URI remains same. We can use @RequestMapping method variable to narrow down the HTTP methods for which this method will be invoked. For example:

```
@RequestMapping(value="/method2", method=RequestMethod.POST)
@ResponseBody
public String method2(){
    return "method2";
}

@RequestMapping(value="/method3",
method={RequestMethod.POST,RequestMethod.GET})
@ResponseBody
public String method3(){
    return "method3";
}
```

31. **@RequestMapping with Headers**: We can specify the headers that should be present to invoke the handler method. For example:

```
@RequestMapping(value="/method4", headers="name=pankaj")
@ResponseBody
public String method4(){
    return "method4";
}

@RequestMapping(value="/method5", headers={"name=pankaj", "id=1"})
@ResponseBody
public String method5(){
    return "method5";
}
```

43. **@RequestMapping with Produces and Consumes**: We can use header Content-Type and Accept to find out request contents and what is the mime message it wants in response. For clarity, @RequestMapping provides **produces** and **consumes** variables where we can specify the request content-type for which method will be invoked and the response content type. For example:

```
@RequestMapping(value="/method6",
produces={"application/json","application/xml"}, consumes="text/html")
@ResponseBody
public String method6(){
    return "method6";
}
```

Above method can consume message only with **Content-Type as text/html** and is able to produce messages of type **application/json** and **application/xml**.

Spring @PathVariable

49. **@RequestMapping with @PathVariable**: RequestMapping annotation can be used to handle dynamic URIs where one or more of the URI value works as a parameter. We can even specify Regular Expression for URI dynamic parameter to accept only specific type of input. It works with **@PathVariable annotation** through which we can map the URI variable to one of the method arguments. For example:

```
@RequestMapping(value="/method7/{id}")
@ResponseBody
public String method7(@PathVariable("id") int id){
    return "method7 with id="+id;
}

@RequestMapping(value="/method8/{id:[\\d]+}/{name}")
@ResponseBody
public String method8(@PathVariable("id") long id, @PathVariable("name") String name){
    return "method8 with id= "+id+" and name="+name;
}
```

Spring @RequestParam

61. **@RequestMapping with @RequestParam for URL parameters**: Sometimes we get parameters in the request URL, mostly in GET requests. We can use @RequestMapping with **@RequestParam annotation** to retrieve the URL parameter and map it to the method argument. For example:

```
@RequestMapping(value="/method9")
@ResponseBody
public String method9(@RequestParam("id") int id){
    return "method9 with id= "+id;
}
```

For this method to work, the parameter name should be “id” and it should be of type int.

67. **@RequestMapping default method**: If value is empty for a method, it works as default method for the controller class. For example:

```
@RequestMapping()
@ResponseBody
public String defaultMethod(){
    return "default method";
}
```

As you have seen above that we have mapped /home to HomeController, this method will be used for the default URI requests.

73. **@RequestMapping fallback method**: We can create a fallback method for the controller class to make sure we are catching all the client requests even though there are no matching handler methods. It is useful in sending custom 404 response pages to users when there are no handler methods for the request.

```
@RequestMapping("*")
@ResponseBody
public String fallbackMethod(){
    return "fallback method";
}
```

What is the difference between PUT, POST and PATCH for RESTful APIs?

POST is used to *create* a *new* entity. “Post” means “after”; if you have a collection of entities and you tack a new one onto its end, you have *posted* to the collection. You can’t post an existing entity, and it’s common (though not always required) to use the collection’s URI to post.

A HTTP.POST method always creates a new resource on the server. Its a non-idempotent request i.e. if user hits same requests 2 times it would create another new resource if there is no constraint.

http post method **is** like a INSERT query **in** SQL which always creates a **new** record **in** database.
Example: Use POST method to save new user, order etc where backend server decides the resource id for new resource.

PATCH is used to *update* an *existing* entity with new information. You can’t patch an entity that doesn’t exist. You would use this when you have a simple update to perform, e.g. changing a user’s name.

A HTTP.PATCH method is used for partial modifications to a resource i.e. delta updates.

http patch method **is** like a UPDATE query **in** SQL which sets **or** updates selected columns only **and not** the whole row.
Example: You could use PATCH method to update order status.

PATCH /api/users/40450236/order/10234557

PUT is used to *set* an entity’s information completely. PUTting is similar to POSTing, except that it will overwrite the entity if already exists or create it otherwise. You could use a PUT to write a user to your database that may already be in it.

In HTTP.PUT method the resource is first identified from the URL and if it exists then it is updated otherwise a new resource is created. When the target resource exists it overwrites that resource with a complete new body. That is HTTP.PUT method is used to CREATE or UPDATE a resource.

http put method **is** like a MERGE query **in** SQL which inserts **or** updates a record depending upon whether the given record exists.

PUT request is idempotent i.e. hitting the same requests twice would update the existing recording (No new record created). In PUT method the resource id is decided by the client and provided in the request URL.

Example: Use PUT method to update existing user or order.