FREE eBook

LEARNING jersey

Free unaffiliated eBook created from
Stack Overflow contributors.

#jersey

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: jersey

It is an unofficial and free jersey ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official jersey.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with jersey

Remarks

This section provides an overview of what jersey is, and why a developer might want to use it.

It should also mention any large subjects within jersey, and link out to the related topics. Since the Documentation for jersey is new, you may need to create initial versions of those related topics.

Examples

Installation or Setup

primary requirement is that java should be installed in your system.there is two option for setting the jersey in the Eclipse IDE is first manually download the jersey jars from this link.and the in the project->add external jars you can add these libraries there.[
https://jersey.java.net/download.html][1]

and second option is through maven you have to add the maven dependency for jersey jars and it will automatically download for you.

```
<dependency>
    <groupId>org.glassfish.jersey.containers</groupId>
    <artifactId>jersey-container-servlet-core</artifactId>
    <version>2.6</version>
</dependency>
```

Hello World Example

this is the simple example of getting the hello world plain text message as output on calling the GET request.

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
@Path("/hello")
public class HelloExample {
   @GET
   @Produces(MediaType.APPLICATION_TEXT)
   public String getUsers(){
      return "Hello World";
   }
}
```

you also nedd to add the following in the web.xml file to completely setup the api.

```
   <display-name>User Message</display-name>
```

```
   <servlet>
      <servlet-name>Jersey REST Api</servlet-name>
      <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servletclass>
      <init-param>
         <param-name>jersey.config.server.provider.packages</param-name>
         <param-value>your_package_name</param-value>
      </init-param>
   </servlet>
   <servlet-mapping>
      <servlet-name>Jersey REST Api</servlet-name>
      <url-pattern>/rest/*</url-pattern>
   </servlet-mapping>
```

After that you will have to deploy this on your server and then open the following Url in your browser to get the output. your_server_name/your_appl_name/rest/hello.

CRUD Operations example in Jersey

This example demonstrates the use of GET, POST, PUT and DELETE HTTP Methods in doing CRUD operations on a REST resource

I am using the below software, frameworks and tools:

1. Jersey 2.25.1
2. JDK 1.7.x (Java 7)
3. Eclipse IDE Kepler
4. Apache Maven 3.3.9
5. Apache Tomcat 7.x

Please follow the below steps for Creating the required Jersey Application

Step 1 : Create a new maven project using *maven-archetype-webapp* archetype in Eclipse IDE by choosing File->New->Maven Project

Step 2 : Add the below dependencies in the project's pom.xml file.

```
<dependencies>
    <dependency>
        <groupId>org.glassfish.jersey.containers</groupId>
        <!-- if your container implements Servlet API older than 3.0, use "jersey-container-
servlet-core" -->
        <artifactId>jersey-container-servlet-core</artifactId>
        <version>2.25.1</version>
    </dependency>
    <dependency>
        <groupId>org.glassfish.jersey.media</groupId>
        <artifactId>jersey-media-jaxb</artifactId>
        <version>2.25.1</version>
    </dependency>
    <dependency>
        <groupId>org.glassfish.jersey.media</groupId>
        <artifactId>jersey-media-json-jackson</artifactId>
        <version>2.25.1</version>
    </dependency>
    <dependency>
```

```
        <groupId>javax.servlet</groupId>
        <artifactId>servlet-api</artifactId>
        <version>2.5</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>1.7.25</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-simple</artifactId>
        <version>1.7.25</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Step 3 : Application Resource Configuration

Create a class extending **org.glassfish.jersey.server.ResourceConfig** class and register the JAX-RS components in its constructor. Here we are registering all the resources under com.stackoverflow.ws.rest package.

```
package com.stackoverflow.ws.rest;

import org.glassfish.jersey.server.ResourceConfig;

public class MyApplication extends ResourceConfig {

    public MyApplication() {
        packages("com.stackoverflow.ws.rest");
    }
}
```

Step 4 : Create a simple Java bean like Employee with properties like id and name. And override the equals() and hashCode() method. Also, the class should have a no argument public constructor. Please find the code below:

Employee Java bean class

```
package com.stackoverflow.ws.rest.model;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Employee {

private int id;
private String name;
```

```
public Employee(){
    super();
}

public Employee(int id, String name) {
    super();
    this.id = id;
    this.name = name;
}

@XmlElement
public int getId() {
    return id;
}
public void setId(int id) {
    this.id = id;
}

@XmlElement
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + id;
    result = prime * result + ((name == null) ? 0 : name.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null) {
        return false;
    }
    if (!(obj instanceof Employee)) {
        return false;
    }
    Employee other = (Employee) obj;
    if (id != other.id) {
        return false;
    }
    if (name == null) {
        if (other.name != null) {
            return false;
        }
    } else if (!name.equals(other.name)) {
        return false;
    }
    return true;
}
}
```

Some additional info on the code

1. Annotations @XmlRootElement and @XmlElement are required for JAXB to marshall and unmarshall the request and response messages.

Step 5 : Create the Employee Resource as given below:

EmployeeResource service class

```
package com.stackoverflow.ws.rest;

import java.net.URI;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.GenericEntity;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriBuilder;
import javax.ws.rs.core.UriInfo;

import com.stackoverflow.ws.rest.model.Employee;

@Path("/employees")
public class EmployeeResource {

    private static Map<Integer, Employee> employeesRepository = new HashMap<Integer,
Employee>();

// Read - get all the employees
@GET
@Produces({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
public Response getEmployees() {
    List<Employee> employees = new ArrayList<Employee>(
            employeesRepository.values());
    GenericEntity<List<Employee>> entity = new GenericEntity<List<Employee>>(
            employees) {
    };
    return Response.ok(entity).build();
}

// Read - get an employee for the given ID
@GET
@Path("/{key}")
@Produces({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
public Response getEmployee(@PathParam("key") int key) {
```

```
    if (employeesRepository.containsKey(key)) {

        return Response.ok(employeesRepository.get(key)).build();
    } else {

        return Response.status(404).build();
    }
}

// Create - create an employee
@POST
@Consumes({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
public Response addEmployee(Employee employee, @Context UriInfo uriInfo) {

    if(employee.getId()!=0){

        return Response.status(400).build();
    }

    int createdEmployeeId = 1;

    if(!employeesRepository.isEmpty()){

        createdEmployeeId = Collections.max(employeesRepository.keySet()) + 1;
    }

    employee.setId(createdEmployeeId);
    employeesRepository.put(createdEmployeeId, employee);

    UriBuilder builder = uriInfo.getAbsolutePathBuilder();
    URI createdURI = builder.path(Integer.toString(createdEmployeeId)).build();
    return Response.created(createdURI).build();
}

// Update - updates an existing employee
@PUT
@Path("/{key}")
@Consumes({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
public Response updateEmployee(@PathParam("key") int key, Employee employee) {

    int status = 0;

    if (employeesRepository.containsKey(key)) {
        // update employeeRepostory
        employeesRepository.put(key, employee);
        status = 204;
    } else {
        status = 404;
    }
    return Response.status(status).build();
}

// Delete - deletes an existing employee
@DELETE
@Path("/{key}")
public Response deleteEmployee(@PathParam("key") int key) {

    employeesRepository.remove(key);
    return Response.noContent().build();
}
```

```
// Delete - deletes all the employees
@DELETE
public Response deleteEmployees() {

    employeesRepository.clear();
    return Response.noContent().build();
}
}
```

Note: Although both POST and PUT method can be used for creating and/or updating a resource, here we are restricting POST method from updating an existing resource and PUT method from creating a new resource. But to know more about the use of these methods please go to this link

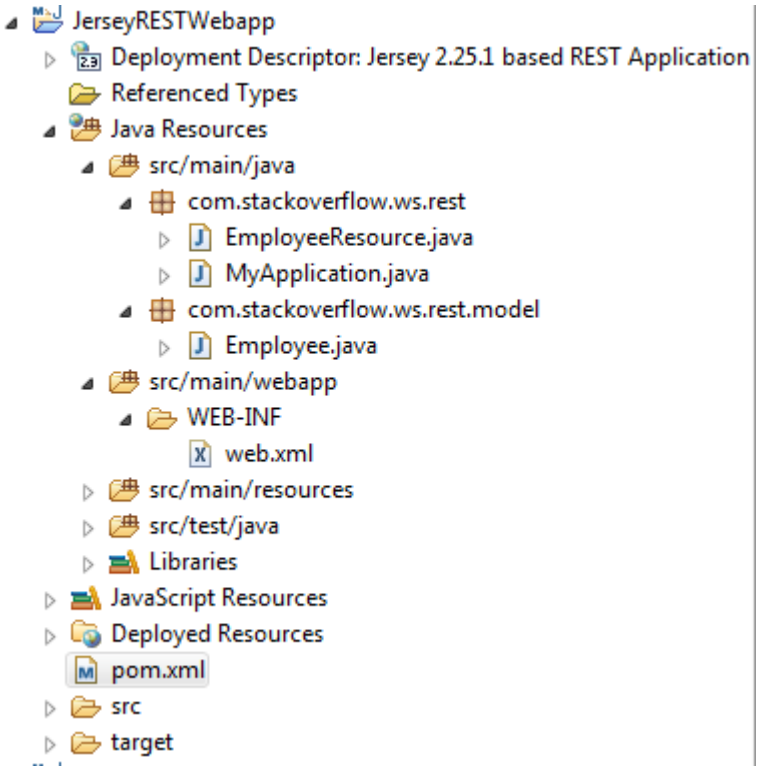
Step 6 : Finally add Jersey Servlet configuration in Deployment Descriptor file (web.xml)

```
<web-app>
  <display-name>Jersey 2.25.1 based REST Application</display-name>

   <servlet>
        <servlet-name>JerseyFrontController</servlet-name>
        <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>com.stackoverflow.ws.rest.MyApplication</param-value>
        </init-param>
    </servlet>

    <servlet-mapping>
        <servlet-name>JerseyFrontController</servlet-name>
        <url-pattern>/ws/rest/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Step 7 : Clean and maven build the project after ensuring the below folder structure in your project.



Step 8 : Run the application in Apache Tomcat.

Now, use some REST client like POSTMAN extension in chrome or SOAP UI to navigate to http://{hostname}:{portnumber}/{projectName/applicationName}/ws/rest/employees, with appropriate HTTP method and don't forget to add **Accept** header with either **application/json** or **application/xml** as value in the HTTP Request.

Read Getting started with jersey online: https://riptutorial.com/jersey/topic/6926/getting-started-with-jersey

Chapter 2: Configuring JAX-RS in Jersey

Examples

Java Jersey CORS filter for Cross Origin Requests

```
@Provider
public class CORSResponseFilter implements ContainerResponseFilter {

    public void filter(
        ContainerRequestContext requestContext,
        ContainerResponseContext responseContext
    ) throws IOException {
        MultivaluedMap<String, Object> headers = responseContext.getHeaders();
        headers.add("Access-Control-Allow-Origin", "*"); //Allow Access from everywhere
        headers.add("Access-Control-Allow-Methods", "GET, POST, DELETE, PUT");
        headers.add("Access-Control-Allow-Headers", "X-Requested-With, Content-Type");
    }
}
```

Note that the Access-Control-Allow-Origin is only useful at OPTIONS responses.

Java Jersey Configuration

This example illustrates how to configure Jersey so that you can begin using it as a JAX-RS implementation framework for your RESTful API.

Assuming that you have already installed Apache Maven, follow these steps to set up Jersey:

1. Create maven web project structure, in terminal (windows) execute the following command

   mvn archetype:generate -DgroupId= com.stackoverflow.rest -DartifactId= jersey-ws-demo -DarchetypeArtifactId=maven-archetype-webapp -DinteractiveMode=false

Note: To support Eclipse, use Maven command : **mvn eclipse:eclipse -Dwtpversion=2.0**

2. Go to the folder where you created your maven project,in your pom.xml, add the required dependencies

```
<dependencies>
    <!-- Jersey 2.22.2 -->
    <dependency>
        <groupId>org.glassfish.jersey.containers</groupId>
        <artifactId>jersey-container-servlet</artifactId>
        <version>${jersey.version}</version>
    </dependency>
    <!-- JSON/POJO support -->
    <dependency>
        <groupId>org.glassfish.jersey.media</groupId>
        <artifactId>jersey-media-json-jackson</artifactId>
        <version>${jersey.version}</version>
```

```
    </dependency>
</dependencies>

<properties>
    <jersey.version>2.22.2</jersey.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
```

3. In Web.xml, add the following code

```
<servlet>
    <servlet-name>jersey-serlvet</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
    <init-param>
        <param-name>jersey.config.server.provider.packages</param-name>
        <!-- Service or resources to be placed in the following package -->
        <param-value>com.stackoverflow.service</param-value>
    </init-param>

    <!-- Application configuration, used for registering resources like filters  -->
    <init-param>
        <param-name>javax.ws.rs.Application</param-name>
        <param-value>com.stackoverflow.config.ApplicationConfig</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
 </servlet>

 <!-- Url mapping, usage-http://domainname:port/appname/api/ -->
 <servlet-mapping>
     <servlet-name>jersey-serlvet</servlet-name>
     <url-pattern>/api/*</url-pattern>
 </servlet-mapping>
```

4. The `ApplicationConfig` class

```
public class ApplicationConfig extends ResourceConfig {
    public ApplicationConfig() {
        register(OtherStuffIfNeeded.class);
    }
}
```

It should also be noted that if you want to go with *no* web.xml, you could simply get rid of it, and add `@ApplicationPath("/api")` on top of the `ApplicationConfig` class.

```
@ApplicationPath("/api")
public class ApplicationConfig extends ResourceConfig {
    public ApplicationConfig() {
        // this call has the same effect as
        // jersey.config.server.provider.packages
        // in the web.xml: it scans that packages for resources and providers.
        packages("com.stackoverflow.service");
    }
}
```

5. Build and deploy your maven project.
6. You can now set up your Java RESTful webservice (JAX-RS) classes to use Jersey's jars.

Read Configuring JAX-RS in Jersey online: https://riptutorial.com/jersey/topic/7012/configuring-jax-rs-in-jersey

Chapter 3: Dependency Injection with Jersey

Examples

Basic Dependency Injection using Jersey's HK2

Jersey (2) uses HK2 as its dependency injection (DI) system. We can use other injection systems, but its infrastructure is built with HK2, and allows us to also use it within our applications.

Setting up simple dependency injection with Jersey takes just a few lines of code. Let say for example we have a service we would like to inject into our resources.

```
public class GreetingService {
    public String getGreeting(String name) {
        return "Hello " + name + "!";
    }
}
```

And we want to inject this service into a Jersey resource

```
@Path("greeting")
public class GreetingResource {

    @Inject
    public GreetingService greetingService;

    @GET
    public String get(@QueryParam("name") String name) {
        return this.greetingService.getGreeting(name);
    }
}
```

In order for the injection to work, all we need is a simple configuration

```
@ApplicationPath("/api")
public class AppConfig extends ResourceConfig {
    public AppConfig() {
        register(GreetingResource.class);
        register(new AbstractBinder() {
            @Override
            protected void configure() {
                bindAsContract(GreetingService.class);
            }
        });
    }
}
```

Here we are saying that we want to bind the `GreetingService` to the injection system, and advertise it as injectable by the same class. What the last statement means is that we can only inject it as `GreetingService` and (probably obviously) not by any other class. As you will see later, it is possible to change this.

That's it. That all you need. If you are not familiar with this `ResourceConfig` configuration (maybe you are using web.xml), pleas seee the Configuring JAX-RS in Jersey topic on SO Docs.

> **Note:** The injection above is field injection, where the service is injected into the field of the resource. Another type of injection is constructor injection, where the service is injected into the constructor

```
private final GreetingService greetingService;

@Inject
public GreetingResource(GreetingService greetingService) {
     this.greetingService = greetingService;
}
```

> This is probably the preferred way to go as opposed to field injection, as it makes the resource easier to unit test. Constructor injection doesn't require any different configuration.

Ok now lets say that instead of a class, the `GreetingService` is an interface, and we have an implementation of it (which is very common). To configure that, we would use the following syntax in the above `configure` method

```
@Override
protected void configure() {
    bind(NiceGreetingService.class).to(GreetingService.class);
}
```

This reads as "bind `NiceGreetingService`, and advertise it as `GreetingService`". This means we can use the exact same code in the `GreetingResource` above, because we advertise the contract as `GreetingService` and not `NiceGreetingService`. But the actual implementation, when injected, will be the `NiceGreetingService`.

Now what about scope. If you've ever worked with any injection framework, you will have came across the concept of scope, which determines the lifespan of the service. You may have heard of a "Request Scope", where the service is alive only for the life of the request. Or a "Singleton Scope", where there is only one instance of the service. We can configure these scopes also using the following syntax.

```
@Override
protected void configure() {
    bind(NiceGreetingService.class)
            .to(GreetingService.class)
            .in(RequestScoped.class);
}
```

The default scope is `PerLookup`, which means that every time this service is requested, a new one will be created. In the example above, using the `RequestScoped`, a new service will be created for a single request. This may or may not be the same as the `PerLookup`, depending on how many places we are trying to inject it. We may be trying to inject it into a filter and into a resource. If this were

`PerLookup`, then two instances would be created for each request. In this case, we only want one.

The other two scopes available are `Singleton` (only one instance created) and `Immediate` (like `Singleton`) but is created on startup (whereas with `Singleton`, it's not created until the first request).

Aside from binding classes, we could also just use an instance. This would gives us a default singleton, so we don need to use the `in` syntax.

```
@Override
protected void configure() {
    bind(new NiceGreetingService())
            .to(GreetingService.class);
}
```

What if we have some complex creation logic or need some request context information for the service. In this case there are `Factory`s. Most things we can inject into our Jersey resources, we can also inject into a `Factory`. Take for example

```
public class GreetingServiceFactory implements Factory<GreetingService> {

    @Context
    UriInfo uriInfo;

    @Override
    public GreetingService provide() {
        return new GreetingService(
                uriInfo.getQueryParameters().getFirst("name"));
    }

    @Override
    public void dispose(GreetingService service) {
        /* noop */
    }
}
```

Here we have a factory, that gets request information from the `UriInfo`, in this case a query parameters, and we create the `GreetingService` from it. To configure it, we use the following syntax

```
@Override
protected void configure() {
    bindFactory(GreetingServiceFactory.class)
            .to(GreetingService.class)
            .in(RequestScoped.class);
}
```

That's it. These are just the basics. There's a lot more things HK and Jersey to do.

Read Dependency Injection with Jersey online:
https://riptutorial.com/jersey/topic/7016/dependency-injection-with-jersey

Chapter 4: Jersey MVC Support

Introduction

MVC Frameworks such as Spring MVC are using to create web applications that serve dynamic web pages. Jersey, though known to be a REST Framework, also has support for create dynamic web pages using its MVC module.

Examples

Jersey MVC Hello World

To get started, create a new Maven webapp (how to do this is outside the scope of this example). In your pom.xml, add the following two dependencies

```
<dependency>
    <groupId>org.glassfish.jersey.containers</groupId>
    <artifactId>jersey-container-servlet</artifactId>
    <version>2.25.1</version>
</dependency>
<dependency>
    <groupId>org.glassfish.jersey.ext</groupId>
    <artifactId>jersey-mvc-jsp</artifactId>
    <version>2.25.1</version>
</dependency>
```

Also in the pom, add the `jetty-maven-plugin` that we'll be using the run the application during development

```
<build>
    <finalName>jersey-mvc-hello-world</finalName>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>2.5.1</version>
            <inherited>true</inherited>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>

        <plugin>
            <groupId>org.eclipse.jetty</groupId>
            <artifactId>jetty-maven-plugin</artifactId>
            <version>9.3.8.v20160314</version>
        </plugin>
    </plugins>
</build>
```

Now we can create our controllers. In any MVC framework, the concepts are usually the same. You have a template, and you use a controller to populate a model that will be used to render the template. The term "render" here is used to mean create the final HTML page by combining the template and the model Take for example this template

src/main/webapp/WEB-INF/jsp/index.jsp

```
<html>
    <head>
        <title>JSP Page</title>
    </head>
    <body>
        <h1>${it.hello} ${it.world}</h1>
    </body>
</html>
```

This is a JSP file. JSP is just one of the template engines supported by Jersey. Here we are using two model variables, `hello`, and `world`. It is expected that these two variables will be in the model that is used to render this template. So let's add the controller

```
package com.example.controller;

import org.glassfish.jersey.server.mvc.Viewable;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import java.util.HashMap;
import java.util.Map;

@Path("/")
public class HomeController {

    @GET
    @Produces(MediaType.TEXT_HTML)
    public Viewable index() {
        Map<String, String> model = new HashMap<>();
        model.put("hello", "Hello");
        model.put("world", "World");
        return new Viewable("/index", model);
    }
}
```

You can see here we're populating the model with the properties `hello` and `world`. Also the controller method returns the name of the view template that is to be used, in this case `index`. With this the framework knows to grab the "index" template, and use the model provided to render it.

Now we just need to configure it. Add a `ResourceConfig` subclass with the following

```
package com.example;

import org.glassfish.jersey.server.ResourceConfig;
import org.glassfish.jersey.server.mvc.jsp.JspMvcFeature;
```

```
public class AppConfig extends ResourceConfig {

    public AppConfig() {
        packages("com.example.controller");
        property(JspMvcFeature.TEMPLATE_BASE_PATH, "/WEB-INF/jsp");
        register(JspMvcFeature.class);
    }
}
```

There are three things going on here:

1. We use `packages` to tell Jersey to scan the `com.example.controller` package for classes annotated with `@Path` to that it can register it. In this case, it registers our `HomeController`.

2. We are setting the base path for the framework to resolve templates. In this case we are telling Jersey to look in the `WEB-INF/jsp` for templates. You can see the `index.jsp` example above in the in this director. Also in the controller we return just the template name `index`. This will be used to find the template, by prefixing the configure base path, and suffixing an implicit `.jsp`

3. We need to register the feature that handles JSP rendering. As mentioned previously, JSP is not the only rendering engine supported by Jersey. There are a couple more supported out of the box.

The last thing we need to do is configure Jersey in the web.xml

```
<filter>
    <filter-name>Jersey</filter-name>
    <filter-class>org.glassfish.jersey.servlet.ServletContainer</filter-class>
    <init-param>
        <param-name>javax.ws.rs.Application</param-name>
        <param-value>com.example.AppConfig</param-value>
    </init-param>
</filter>

<filter-mapping>
    <url-pattern>/*</url-pattern>
    <filter-name>Jersey</filter-name>
</filter-mapping>
```

Here we are just configuring Jersey to use our `AppConfig` class. One very important thing to point out here, is the use of the `<filter>` instead of what you would normally see, a `<servlet>`. This is required when using JSP as the template engine.

Now we can run it. From the command line run `mvn jetty:run`. This will run the Maven Jetty plugin we configured previously. When you see "Started Jetty Server", the server is ready. Go to the browser URL `http://localhost:8080/`. Voila, "Hello World". Enjoy.

For more information, see the Jersey Documentation for MVC Templates

Read Jersey MVC Support online: https://riptutorial.com/jersey/topic/9989/jersey-mvc-support

Chapter 5: Using Spring Boot with Jersey

Examples

Simple Application with Spring Boot and Jersey

Spring Boot is a bootstrapping framework for Spring applications. It has seamless support for integrating with Jersey also. One of the advantages of this (from the perspective of a Jersey user), is that you have access to Spring's vast ecosystem.

To get started, create a new *standalone* (non-wepapp) Maven project. We can create a webapp also, but for this guide, we will just use a standalone app. Once you've create the project, add the following to your `pom.xml`

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
</properties>
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.0.RELEASE</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jersey</artifactId>
    </dependency>
    <dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-tomcat</artifactId>
    </dependency>
</dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

We just need two dependencies. One for the Jersey Spring Boot module, and another for an embedded Tomcat server. The plugin we will use to run the application to test.

Once you have that, add the following classes to the project.

```
com/example
  |
  +-- GreetingApplication.class
  +-- JerseyConfig.class
  |
```

```
  + com/example/services
  |   |
  |   +-- GreetingService.class
  |   +-- NiceGreetingService.class
  |
  + com/examples/resources
      |
      +-- GreetingResource.class
```

`GreetingApplication.class`

This is the bootstrap class (very simple)

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class GreetingApplication {

    public static void main(String[] args) {
        SpringApplication.run(GreetingApplication.class, args);
    }
}
```

`JerseyConfig.class`

This is the Jersey configuration class

```
import javax.ws.rs.ApplicationPath;
import org.glassfish.jersey.server.ResourceConfig;
import org.springframework.stereotype.Component;

@Component
@ApplicationPath("/api")
public class JerseyConfig extends ResourceConfig {
    public JerseyConfig() {
        packages("com.example");
    }
}
```

`GreetingService.class` and **`NiceGreetingService.class`**

```
public interface GreetingService {
   public String getGreeting(String name);
}

import org.springframework.stereotype.Component;

@Component
public class NiceGreetingService implements GreetingService {

    @Override
    public String getGreeting(String name) {
        return "Hello " + name + "!";
    }
}
```

GreetingResource

This is the resource class where we will let Spring inject the `GreetingService` into.

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.QueryParam;
import org.springframework.beans.factory.annotation.Autowired;
import com.example.service.GreetingService;

@Path("greeting")
public class GreetingResource {

    private GreetingService greetingService;

    @Autowired
    public GreetingResource(GreetingService greetingService) {
        this.greetingService = greetingService;
    }

    @GET
    public String getGreeting(@QueryParam("name") String name) {
        return this.greetingService.getGreeting(name);
    }
}
```

And that's it. We can now run the application. Grab a terminal and run the following command from the root of the project.

```
mvn spring-boot:run
```

The application should take a few seconds to get started. There will be some logging, and you will see some Spring ASCII art. After that art, it should be about 30 lines or so of more logging, then you should see

```
 15.784 seconds (JVM running for 38.056)
```

Now the app is started. If you use cURL you can test it with

```
curl -v 'http://localhost:8080/api/greeting?name=peeskillet'
```

If you are on Windows, use double quotes around the URL. If you aren't using cURL, just type it in the browser. You should see the result

```
Hello peeskillet!
```

You may notice the request takes a few seconds on the first request you make. That is because Jersey is not fully loaded when the app launches. We can change that by adding a `application.properties` file in the `src/main/resources` folder. In that file add the following:

```
spring.jersey.servlet.load-on-startup=1
```

Read Using Spring Boot with Jersey online: https://riptutorial.com/jersey/topic/7019/using-spring-boot-with-jersey

Credits

S. No	Chapters	Contributors
1	Getting started with jersey	Community, Praveen, Rednivrug
2	Configuring JAX-RS in Jersey	BALAJI RAJ, peeskillet, Sampada, Stephen C, Tobias Friedinger
3	Dependency Injection with Jersey	fujy, peeskillet, zyexal
4	Jersey MVC Support	peeskillet
5	Using Spring Boot with Jersey	peeskillet, pzaenger