EBook Gratis

# APRENDIZAJE

# jersey

Free unaffiliated eBook created from
**Stack Overflow contributors.**

#jersey

# Tabla de contenido

# Acerca de

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

# Capítulo 1: Empezando con jersey

## Observaciones

Esta sección proporciona una descripción general de qué es una camiseta y por qué un desarrollador podría querer usarla.

También debe mencionar cualquier tema importante dentro de jersey, y vincular a los temas relacionados. Dado que la Documentación para jersey es nueva, es posible que deba crear versiones iniciales de esos temas relacionados.

## Examples

### Instalación o configuración

El requisito principal es que java se instale en su sistema. Hay dos opciones para configurar la camiseta en el IDE de Eclipse. Primero, descargue manualmente los tarros de la camiseta desde este enlace. Y en el proyecto -> Agregar jarras externas, puede agregar estas bibliotecas. allí [ https://jersey.java.net/download.html◆◆1]

y la segunda opción es a través de maven: tiene que agregar la dependencia de maven para los tarros de jersey y se descargará automáticamente para usted.

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

### Hola mundo ejemplo

Este es el ejemplo simple de obtener el mensaje de texto simple hello world como salida al llamar a la solicitud GET.

```
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";
   }
}
```

También es necesario agregar lo siguiente en el archivo web.xml para configurar completamente

la 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>
```

Después de eso, deberá implementar esto en su servidor y luego abrir la siguiente URL en su navegador para obtener el resultado. su_servidor_nombre / su_appl_nombre / resto / hola.

## Ejemplo de operaciones de CRUD en Jersey

Este ejemplo demuestra el uso de los métodos GET, POST, PUT y DELETE HTTP al realizar operaciones CRUD en un recurso REST

Estoy utilizando el siguiente software, marcos y herramientas:

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

Siga los pasos a continuación para crear la aplicación de Jersey requerida

Paso 1: cree un nuevo proyecto de maven utilizando *el arquetipo maven-archetype-webapp* en Eclipse IDE seleccionando Archivo-> Nuevo-> Proyecto de Maven

Paso 2: Agregue las dependencias a continuación en el archivo pom.xml del proyecto.

```
<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>
```

Paso 3: Configuración de recursos de la aplicación

Cree una clase de extensión **org.glassfish.jersey.server.ResourceConfig** y registre los componentes JAX-RS en su constructor. Aquí estamos registrando todos los recursos en el paquete com.stackoverflow.ws.rest.

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

import org.glassfish.jersey.server.ResourceConfig;

public class MyApplication extends ResourceConfig {

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

Paso 4: Cree un bean Java simple como Employee con propiedades como id y name. Y anule el método equals () y hashCode (). Además, la clase debe tener un constructor público sin argumentos. Por favor encuentre el código a continuación:

Empleado de Java clase de frijol

```
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;
}
}
```

Alguna información adicional sobre el código.

1. Las anotaciones @XmlRootElement y @XmlElement son necesarias para que JAXB marque y elimine todos los mensajes de solicitud y respuesta.

Paso 5: Crea el Recurso del empleado como se indica a continuación:

Clase de servicio EmployeeResource

```
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();
}
}
```

**Nota:** Aunque los métodos POST y PUT se pueden usar para crear y / o actualizar un recurso, aquí estamos restringiendo el método POST de actualizar un recurso existente y el método PUT de crear un nuevo recurso. Pero para saber más sobre el uso de estos métodos, por favor vaya a este enlace

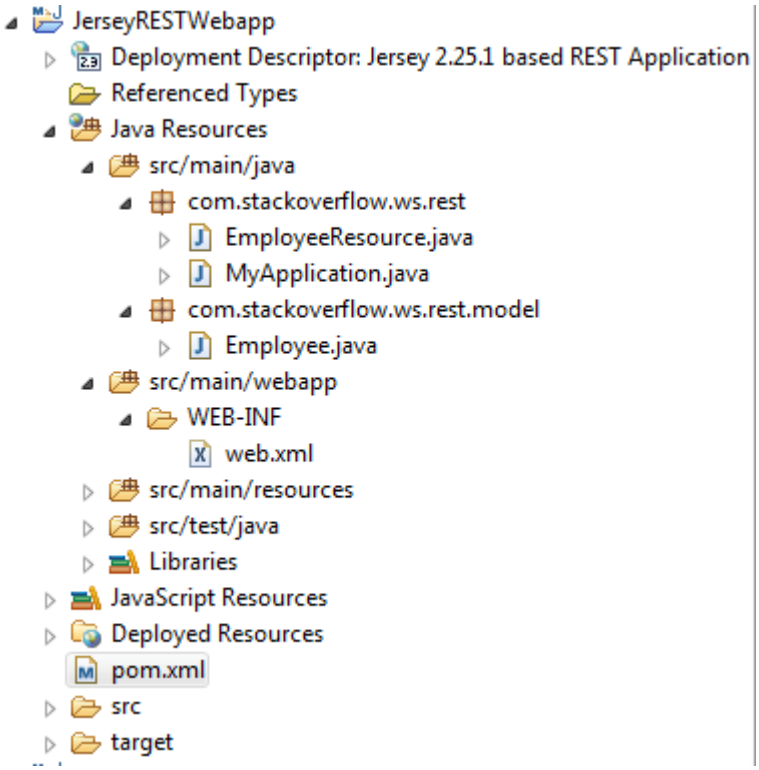
Paso 6: Finalmente, agregue la configuración de Jersey Servlet en el archivo Descriptor de implementación (web.xml)

```
<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>
```

Paso 7: Limpie y cree el proyecto, después de asegurarse de que la estructura de carpetas de su proyecto sea la siguiente.




Paso 8: Ejecutar la aplicación en Apache Tomcat.

Ahora, use algún cliente REST como la extensión POSTMAN en Chrome o SOAP UI para navegar a http: // {hostname}: {número de puerto} / {projectName / applicationName} / ws / rest / employee, con el método HTTP apropiado y no lo olvide para agregar el encabezado **Aceptar** con **application / json** o **application / xml** como valor en la solicitud HTTP.

Lea Empezando con jersey en línea: https://riptutorial.com/es/jersey/topic/6926/empezando-con-jersey

# Capítulo 2: Ayuda de Jersey MVC

## Introducción

MVC Frameworks, como Spring MVC, se utilizan para crear aplicaciones web que sirven páginas web dinámicas. Jersey, aunque se sabe que es un marco REST, también tiene soporte para crear páginas web dinámicas utilizando su módulo MVC.

## Examples

### Jersey MVC Hello World

Para comenzar, cree una nueva aplicación web de Maven (cómo hacerlo está fuera del alcance de este ejemplo). En su pom.xml, agregue las siguientes dos dependencias

```
<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>
```

También en el pom, agregue el `jetty-maven-plugin` que `jetty-maven-plugin` ejecutar la aplicación durante el desarrollo

```
<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>
```

Ahora podemos crear nuestros controladores. En cualquier marco MVC, los conceptos son generalmente los mismos. Tiene una plantilla y utiliza un controlador para rellenar un modelo que se utilizará para representar la plantilla. El término "renderizar" aquí se usa para significar crear la página HTML final combinando la plantilla y el modelo Tome, por ejemplo, esta plantilla

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

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

Este es un archivo JSP. JSP es solo uno de los motores de plantillas soportados por Jersey. Aquí estamos usando dos variables de modelo, `hello` y `world` . Se espera que estas dos variables estén en el modelo que se utiliza para representar esta plantilla. Así que vamos a agregar el controlador

```
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);
    }
}
```

Puedes ver aquí que estamos poblando el modelo con las propiedades `hello` y `world` . Además, el método del controlador devuelve el nombre de la plantilla de vista que se va a utilizar, en este caso `index` . Con esto, el marco sabe que debe tomar la plantilla de "índice" y usar el modelo proporcionado para representarlo.

Ahora solo necesitamos configurarlo. Agregue una subclase `ResourceConfig` con lo siguiente

```
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);
    }
}
```

Hay tres cosas que suceden aquí:

1. Usamos `packages` para decirle a Jersey que escanee el paquete `com.example.controller` para las clases anotadas con `@Path` para que pueda registrarlo. En este caso, registra nuestro `HomeController` .

2. Estamos configurando la ruta base para que el marco resuelva las plantillas. En este caso, le estamos diciendo a Jersey que busque plantillas en `WEB-INF/jsp` . Puede ver el ejemplo de `index.jsp` anterior en este director. También en el controlador devolvemos solo el `index` nombre de la plantilla. Esto se usará para encontrar la plantilla, prefijando la ruta de configuración de la base y colocando un sufijo implícito en `.jsp`

3. Necesitamos registrar la característica que maneja el renderizado JSP. Como se mencionó anteriormente, JSP no es el único motor de renderizado soportado por Jersey. Hay un par más apoyado fuera de la caja.

Lo último que debemos hacer es configurar Jersey en el 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>
```

Aquí solo estamos configurando Jersey para usar nuestra clase `AppConfig` . Una cosa muy importante a destacar aquí es el uso del `<filter>` lugar de lo que normalmente vería, un `<servlet>` . Esto es necesario cuando se utiliza JSP como motor de plantillas.

Ahora podemos ejecutarlo. Desde la línea de comandos ejecuta `mvn jetty:run` . Esto ejecutará el complemento Maven Jetty que configuramos previamente. Cuando vea "Started Jetty Server", el servidor estará listo. Vaya a la URL del navegador `http://localhost:8080/` . Voila, "Hola Mundo". Disfrutar.

Para obtener más información, consulte la documentación de Jersey para plantillas MVC.

Lea Ayuda de Jersey MVC en línea: https://riptutorial.com/es/jersey/topic/9989/ayuda-de-jersey-mvc

# Capítulo 3: Configurando JAX-RS en Jersey

## Examples

### Filtro de Java Jersey CORS para solicitudes de origen cruzado

```
@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");
    }
}
```

Tenga en cuenta que Access-Control-Allow-Origin solo es útil en las respuestas de OPCIONES.

### Configuración de Java Jersey

Este ejemplo ilustra cómo configurar Jersey para que pueda comenzar a usarlo como un marco de implementación JAX-RS para su API RESTful.

Suponiendo que ya ha instalado Apache Maven , siga estos pasos para configurar Jersey:

1. Crear estructura de proyecto web maven, en terminal (windows) ejecute el siguiente comando

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

**Nota:** para admitir Eclipse, use el comando Maven: **mvn eclipse: eclipse -Dwtpversion = 2.0**

2. Vaya a la carpeta donde creó su proyecto de maven, en su pom.xml, agregue las dependencias necesarias

```
<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. En Web.xml, agregue el siguiente código

```
<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. La clase `ApplicationConfig`

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

También se debe tener en cuenta que si desea ir *sin* web.xml, simplemente puede deshacerse de él y agregar `@ApplicationPath("/api")` en la parte superior de la clase `ApplicationConfig` .

```
@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. Construye y despliega tu proyecto maven.

6. Ahora puede configurar sus clases de servicio web RESTful de Java (JAX-RS) para usar los tarros de Jersey.

Lea Configurando JAX-RS en Jersey en línea:
https://riptutorial.com/es/jersey/topic/7012/configurando-jax-rs-en-jersey

# Capítulo 4: Inyección de dependencia con jersey

## Examples

### Inyección de dependencia básica utilizando HK2 de Jersey

Jersey (2) utiliza HK2 como su sistema de inyección de dependencia (DI). Podemos usar otros sistemas de inyección, pero su infraestructura está construida con HK2 y nos permite usarla también dentro de nuestras aplicaciones.

Configurar una inyección de dependencia simple con Jersey requiere solo unas pocas líneas de código. Digamos, por ejemplo, que tenemos un servicio que nos gustaría inyectar en nuestros recursos.

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

Y queremos inyectar este servicio en un recurso de Jersey.

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

    @Inject
    public GreetingService greetingService;

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

Para que la inyección funcione, todo lo que necesitamos es una configuración simple.

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

Aquí estamos diciendo que queremos vincular el `GreetingService` al sistema de inyección y anunciarlo como inyectable por la misma clase. Lo que significa la última declaración es que solo podemos inyectarlo como `GreetingService` y (probablemente obviamente) no por ninguna otra clase. Como verás más adelante, es posible cambiar esto.

Eso es. Que todo lo que necesitas. Si no está familiarizado con esta configuración de `ResourceConfig` (quizás esté usando web.xml), consulte el tema de configuración de JAX-RS en Jersey en SO Docs.

---

> **Nota:** la inyección anterior es la inyección de campo, donde el servicio se inyecta en el campo del recurso. Otro tipo de inyección es la inyección de constructor, donde el servicio se inyecta en el constructor

```
private final GreetingService greetingService;

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

> Esta es probablemente la forma preferida de ir en lugar de la inyección de campo, ya que facilita la prueba unitaria del recurso. La inyección de constructor no requiere ninguna configuración diferente.

---

Ok, ahora digamos que en lugar de una clase, `GreetingService` es una interfaz, y tenemos una implementación de la misma (que es muy común). Para configurar eso, usaríamos la siguiente sintaxis en el método de `configure` anterior

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

Esto se lee como "vincular `NiceGreetingService` , y publicitarlo como `GreetingService` ". Esto significa que podemos usar el mismo código exacto en el `GreetingResource` anterior, porque anunciamos el contrato como `GreetingService` y no como `NiceGreetingService` . Pero la implementación real, cuando se inyecte, será el `NiceGreetingService` .

Ahora qué pasa con el alcance. Si alguna vez ha trabajado con un marco de inyección, se habrá topado con el concepto de alcance, que determina la vida útil del servicio. Es posible que haya oído hablar de un "Ámbito de solicitud", donde el servicio está activo solo durante la vida útil de la solicitud. O un "Alcance de Singleton", donde solo hay una instancia del servicio. Podemos configurar estos ámbitos también utilizando la siguiente sintaxis.

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

```
            .in(RequestScoped.class);
}
```

El alcance predeterminado es `PerLookup` , lo que significa que cada vez que se solicite este servicio, se creará uno nuevo. En el ejemplo anterior, utilizando `RequestScoped` , se `RequestScoped` un nuevo servicio para una sola solicitud. Esto puede o no puede ser lo mismo que el `PerLookup` , dependiendo de cuántos lugares estamos tratando de inyectarlo. Podemos estar intentando inyectarlo en un filtro y en un recurso. Si esto fuera `PerLookup` , entonces se `PerLookup` dos instancias para cada solicitud. En este caso, solo queremos uno.

Los otros dos ámbitos disponibles son `Singleton` (solo se creó una instancia) e `Immediate` (como `Singleton` ) pero se crea en el inicio (mientras que con `Singleton` , no se crea hasta la primera solicitud).

Aparte de las clases vinculantes, también podríamos usar una instancia. Esto nos da un producto único defecto, así que nosotros no necesita usar la `in` sintaxis.

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

¿Qué sucede si tenemos alguna lógica de creación compleja o necesitamos información de contexto de solicitud para el servicio? En este caso hay `Factory` s. La mayoría de las cosas que podemos inyectar en nuestros recursos de Jersey, también podemos inyectarlas en una `Factory` . Tomar como ejemplo

```
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 */
    }
}
```

Aquí tenemos una fábrica, que obtiene información de solicitud de `UriInfo` , en este caso, un parámetro de consulta, y creamos el Servicio de `GreetingService` partir de él. Para configurarlo, utilizamos la siguiente sintaxis.

```
@Override
protected void configure() {
    bindFactory(GreetingServiceFactory.class)
```

```
            .to(GreetingService.class)
            .in(RequestScoped.class);
}
```

Eso es. Estos son sólo los conceptos básicos. Hay muchas más cosas que HK y Jersey tienen que hacer.

Lea Inyección de dependencia con jersey en línea:
https://riptutorial.com/es/jersey/topic/7016/inyeccion-de-dependencia-con-jersey

# Capítulo 5: Usando Spring Boot con Jersey

## Examples

### Aplicación simple con Spring Boot y Jersey

Spring Boot es un marco de arranque para aplicaciones Spring. Tiene soporte sin problemas para la integración con Jersey también. Una de las ventajas de esto (desde la perspectiva de un usuario de Jersey), es que tiene acceso al vasto ecosistema de Spring.

Para comenzar, cree un nuevo proyecto Maven *independiente* (no wepapp). También podemos crear una aplicación web, pero para esta guía, solo usaremos una aplicación independiente. Una vez que haya creado el proyecto, agregue lo siguiente a su `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>
```

Solo necesitamos dos dependencias. Uno para el módulo Jersey Spring Boot y otro para un servidor Tomcat incorporado. El complemento que usaremos para ejecutar la aplicación para probar.

Una vez que tenga eso, agregue las siguientes clases al proyecto.

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

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

**`GreetingApplication.class`**

Esta es la clase bootstrap (muy 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`**

Esta es la clase de configuración de Jersey

```
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`** y **`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**

Esta es la clase de recurso en la que permitiremos a Spring inyectar el Servicio de `GreetingService` .

```
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);
    }
}
```

Y eso es. Ahora podemos ejecutar la aplicación. Toma un terminal y ejecuta el siguiente comando desde la raíz del proyecto.

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

La aplicación debe tardar unos segundos en comenzar. Habrá algunos registros y verá algunas ilustraciones de Spring ASCII. Después de ese arte, debería ser alrededor de 30 líneas o más de registro, entonces debería ver

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

Ahora se inicia la aplicación. Si usas cURL puedes probarlo con

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

Si está en Windows, use comillas dobles alrededor de la URL. Si no está utilizando cURL, simplemente escríbalo en el navegador. Deberías ver el resultado.

```
Hello peeskillet!
```

Es posible que note que la solicitud demora unos segundos en la primera solicitud que realice. Esto se debe a que Jersey no está completamente cargado cuando se lanza la aplicación. Podemos cambiar eso agregando un archivo `application.properties` en la carpeta

`src/main/resources` . En ese archivo agregue lo siguiente:

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

Lea Usando Spring Boot con Jersey en línea: https://riptutorial.com/es/jersey/topic/7019/usando-spring-boot-con-jersey

# Creditos

| S. No | Capítulos | Contributors |
|---|---|---|
| 1 | Empezando con jersey | Community, Praveen, Rednivrug |
| 2 | Ayuda de Jersey MVC | peeskillet |
| 3 | Configurando JAX-RS en Jersey | BALAJI RAJ, peeskillet, Sampada, Stephen C, Tobias Friedinger |
| 4 | Inyección de dependencia con jersey | fujy, peeskillet, zyexal |
| 5 | Usando Spring Boot con Jersey | peeskillet, pzaenger |