**Notas antes del tutorial**

Para validar Maven: ***mvn validate***

Para instalación limpia Maven (si es necesario): ***mvn -U clean install***

Para compilar en un .Jar: ***mvn package***

**Configuración inicial del pom.xml:**

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
  
  
 <groupId>com.marcobehler</groupId>  
 <artifactId>myfancypdfinvoices</artifactId>  
 <version>1.0-SNAPSHOT</version>  
  
 <packaging>jar</packaging>  
  
 <properties>  
 <maven.compiler.source>17</maven.compiler.source>  
 <maven.compiler.target>17</maven.compiler.target>  
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
 </properties>

**Añadiendo dependencia en el pom.xml:**

Establecemos el servidor apache desde el mismo pom, Maven se encargara del resto.

<dependencies>  
 <dependency>  
 <groupId>org.apache.tomcat.embed</groupId>  
 <artifactId>tomcat-embed-core</artifactId>  
 <version>10.0.20</version>  
 </dependency>  
  
</dependencies>

**Añadiendo plugins:**

<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-shade-plugin</artifactId>  
 <version>3.2.4</version>  
 <executions>  
 <execution>  
 <phase>package</phase>  
 <goals>  
 <goal>shade</goal>  
 </goals>  
 <configuration>  
 <transformers>  
 <transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">  
 <mainClass>com.marcobehler.ApplicationLauncher</mainClass>  
 </transformer>  
 </transformers>  
 </configuration>  
 </execution>  
 </executions>  
</plugin>

Esto es la configuración del Maven shade.

<groupId>org.apache.maven.plugins</groupId>  
<artifactId>maven-shade-plugin</artifactId>  
<version>3.2.4</version>

Tan pronto como llame al mvn package, el complemento shade ejecutará su objetivo shade, que se asegura de incluir todas las bibliotecas de terceros dentro de su archivo .jar, además de su propio código fuente.

<execution>  
 <phase>package</phase>  
 <goals>  
 <goal>shade</goal>  
 </goals>

¿Para qué necesita este bloque, ya que simplemente establece la clase principal en el archivo manifest.mf?

<configuration>  
 <transformers>  
 <transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">  
 <mainClass>com.marcobehler.ApplicationLauncher</mainClass>  
 </transformer>  
 </transformers>  
</configuration>

Tiene que ver con la forma en que funciona el plugin de sombra. El plugin de sombra probablemente descomprimirá todas las bibliotecas de terceros y la pondrá en tu archivo .jar. Si una biblioteca de terceros viene con un archivo manifest.mf, posiblemente hay archivos n-manifest entrando en su archivo .jar, sobrescribiéndose entre sí.

Si especifica la clase principal aquí, el complemento de sombra se asegurará de que su archivo de manifiesto final tenga la clase correcta especificada. ¡Esto también significa que puede eliminar la sección maven-jar-plugin que acaba de agregar en el paso anterior! Esto:

<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-jar-plugin</artifactId>  
 <version>3.2.0</version>  
  
 <configuration>  
 <archive>  
 <manifest>  
 <mainClass>com.marcobehler.ApplicationLauncher</mainClass>  
 </manifest>  
 </archive>  
 </configuration>  
  
</plugin>

**Creación de un punto final JSON para las solicitudes GET del navegador**

package com.marcobehler;  
  
import jakarta.servlet.ServletException;  
import jakarta.servlet.http.HttpServlet;  
import jakarta.servlet.http.HttpServletRequest;  
import jakarta.servlet.http.HttpServletResponse;  
  
import java.io.IOException;  
  
public class MyFancyPdfInvoicesServlet extends HttpServlet {  
  
 @Override  
 protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  
 if (request.getRequestURI().equalsIgnoreCase("/")) {  
 response.setContentType("text/html; charset=UTF-8");  
 response.getWriter().print(  
 "<html>\n" +  
 "<body>\n" +  
 "<h1>Hello World</h1>\n" +  
 "<p>This is my very first, embedded Tomcat, HTML Page!</p>\n" +  
 "</body>\n" +  
 "</html>");  
 }  
 else if (request.getRequestURI().equalsIgnoreCase("/invoices")) {  
 response.setContentType("application/json; charset=UTF-8");  
 response.getWriter().print("[]");  
 }  
 }  
}

MyFancyPdfInvoicesServlet es ahora el nombre del servlet.

if (request.getRequestURI().equalsIgnoreCase("/")) {

…

}  
else if (request.getRequestURI().equalsIgnoreCase("/invoices")) {  
 …  
}

El servlet ahora verifica el URI de solicitud entrante y muestra una página HTML para **/** y devuelve JSON para **/invoices** (facturas).

else if (request.getRequestURI().equalsIgnoreCase("/invoices")) {  
 response.setContentType("application/json; charset=UTF-8");  
 response.getWriter().print("[]");  
}

devolvemos JSON estableciendo el tipo de contenido apropiado de la respuesta HTTP a la *application/json*, haciendo saber al navegador que, de hecho, le está enviando JSON.

Y la forma más sencilla de enviar JSON, por ahora, es devolver una matriz JSON vacía como una cadena simple:*[]* . Es un gran punto de control para ver si todo está funcionando, antes de empezar a tener facturas reales en tu sistema.

#### Punto de control: Creación de un punto final JSON para las solicitudes GET del navegador

#### Reiniciaremos la aplicación (es decir, el método principal de su ApplicationLauncher) y navegaremos a [*http://localhost:8080/invoices*](http://localhost:8080/invoices)*.*

**Creamos la class Factura (Invoice).**

package com.marcobehler;  
import java.util.UUID;  
  
public class Invoice {  
  
 private String id, userId, pdfUrl;  
 private Integer amount;  
  
 // constructor por defecto  
 public Invoice() {  
 }  
  
 // constructor  
 public Invoice(String userId, Integer amount, String pdfUrl) {  
 this.id = UUID.*randomUUID*().toString();  
 this.userId = userId;  
 this.pdfUrl = pdfUrl;  
 this.amount = amount;  
 }  
  
 // getters and setters  
 public String getId() {  
 return id;  
 }  
 public void setId(String id) {  
 this.id = id;  
 }  
 public String getUserId() {  
 return userId;  
 }  
 public void setUserId(String userId) {  
 this.userId = userId;  
 }  
 public String getPdfUrl() {  
 return pdfUrl;  
 }  
 public void setPdfUrl(String pdfUrl) {  
 this.pdfUrl = pdfUrl;  
 }  
 public Integer getAmount() {  
 return amount;  
 }  
 public void setAmount(Integer amount) {  
 this.amount = amount;  
 }  
}

#### Creación de una clase InvoiceService

La creación real de facturas va a ocurrir en un *InvoiceService*, que tendrá este aspecto:

public class InvoiceService {  
 public Invoice create(String userId, Integer amount) {  
 // *TODO real pdf creation and storing it on network server* return new Invoice(userId, amount, "http://www.africau.edu/images/default/sample.pdf");  
 }  
}

**InvoiceService** tiene un método de creación que toma dos parámetros, **el ID de usuario y la cantidad**.

#### Biblioteca JSON de terceros

Tenemos la clase de dominio ***Invoice***, así como el ***InvoiceService***. Ahora, tenemos que dar al servlet capacidades JSON. Para ello, es necesario añadir una biblioteca JSON de terceros al proyecto.

Una de las opciones más populares en el ecosistema Java para hacer conversiones JSON se llama [Jackson](https://github.com/FasterXML/jackson).

#### Hay que añadir la siguiente dependencia a tu pom.xml para añadir la última versión de Jackson al proyecto (no te preocupes por la versión exacta, ya que *cualquier* versión de Jackson funcionará).

<dependency>  
 <groupId>com.fasterxml.jackson.core</groupId>  
 <artifactId>jackson-databind</artifactId>  
 <version>2.13.0</version>  
</dependency>

#### Además del método doGet de antes, ahora también se necesita reescribir (@Override) el método doPost para manejar las solicitudes POST.

@Override  
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws IOException {  
 if (request.getRequestURI().equalsIgnoreCase("/invoices")) {  
  
 String userId = request.getParameter("user\_id");  
 Integer amount = Integer.*valueOf*(request.getParameter("amount"));  
  
 Invoice invoice = new InvoiceService().create(userId, amount);  
  
 response.setContentType("application/json; charset=UTF-8");  
 String json = new ObjectMapper().writeValueAsString(invoice);  
 response.getWriter().print(json);  
 } else {  
 response.setStatus(HttpServletResponse.*SC\_NOT\_FOUND*);  
 }  
}

Al añadir la dependencia si hay error en el auto-import de la class ObjectMapper(), se debería de Invalidar la caches del IDE (File->Invalidate Caches…), de esta forma podremos hacer el import necesario, otra opción es conocerlo de antemano:

import com.fasterxml.jackson.databind.ObjectMapper;

#### Punto de control: Publicar en su punto final JSON

#### Ahora abriremos Postman por ejemplo, y con la aplicación corriendo pondremos lo siguiente:

[*http://localhost:8080/invoices?user\_id=freddieFox&amount=50*](http://localhost:8080/invoices?user_id=freddieFox&amount=50)

#### Nos debería de salir la siguiente respuesta:

#### 

#### Refactorización de la aplicación

#### El *InvoiceService* hasta ahora solo contiene un método para crear nuevas facturas. Realizaremos un método que encuentre todas las facturas existentes, usaremos una lista sencilla en memoria para almacenar facturas (en lugar de utilizar algo más complejo como una base de datos).

#### Almacena todas las facturas en una lista segura para subprocesos. *CopyOnWriteArrayList* es seguro para subprocesos, mientras que ArrayList no lo sería.

#### Al crear una factura, tenemos que asegurarnos de agregar la nueva factura a la lista de facturas.

#### InvoiceService esta correcto, pero el servlet todavía está devolviendo un JSON vacío y codificado de forma rígida []. Lo cambiaremos de la siguiente manera:

#### Crearemos 2 variables de forma global:

private InvoiceService invoiceService = new InvoiceService();  
private ObjectMapper objectMapper = new ObjectMapper();

#### Y modificaremos el doGet para que nos retorne lo que deseamos:

@Override  
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  
 if (request.getRequestURI().equalsIgnoreCase("/")) {  
 response.setContentType("text/html; charset=UTF-8");  
 response.getWriter().print(  
 "<html>\n" +  
 "<body>\n" +  
 "<h1>Hello World</h1>\n" +  
 "<p>This is my very first, embedded Tomcat, HTML Page!</p>\n" +  
 "</body>\n" +  
 "</html>");  
 } else if (request.getRequestURI().equalsIgnoreCase("/invoices")) {  
 response.setContentType("application/json; charset=UTF-8");  
 List<Invoice> invoices = invoiceService.findAll();  
 response.getWriter().print(objectMapper.writeValueAsString(invoices));   
 }  
}

#### Realmente hemos modificado:

#### Esto.

} else if (request.getRequestURI().equalsIgnoreCase("/invoices")) {  
 response.setContentType("application/json; charset=UTF-8");  
 response.getWriter().print("[]");  
}

#### Por esto otro.

} else if (request.getRequestURI().equalsIgnoreCase("/invoices")) {  
 response.setContentType("application/json; charset=UTF-8");  
 List<Invoice> invoices = invoiceService.findAll();   
 response.getWriter().print(objectMapper.writeValueAsString(invoices));  
}

#### En lugar de escribir una matriz [] JSON codificada de forma rígida, ahora está llamando al servicio de facturas para recuperar todas las facturas y convertirlas a JSON con su objectMapper.

#### En el doPost también efecturaremos dos cambios, ya que hemos declarado las 2 variables globales, no es necesario, ni efectivo hacer el new InvoiceService, ni el new ObjectMapper.

Invoice invoice = invoiceService.create(userId, amount);  
response.setContentType("application/json; charset=UTF-8");  
String json = objectMapper.writeValueAsString(invoice);

#### Punto de control: Servicio REST completo

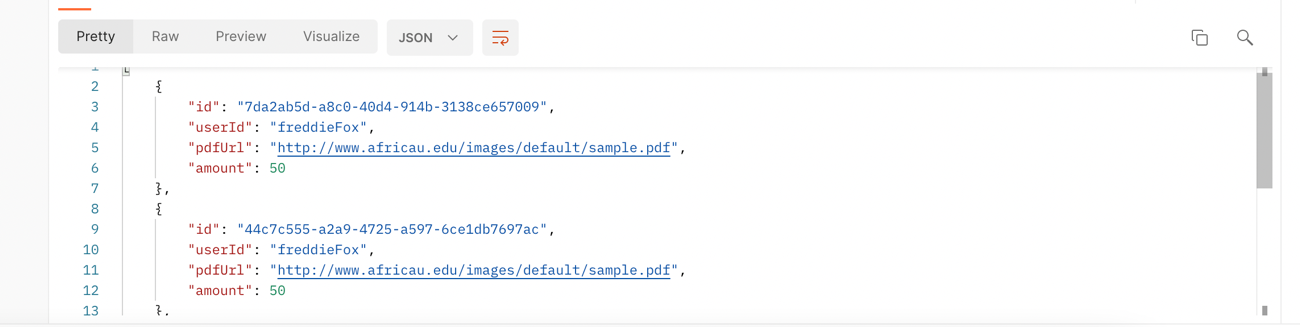
#### Reiniciaremos la aplicación y abriremos Postman, ejecutaremos 2 veces:

POST http://localhost:8080/invoices?user\_id=freddieFox&amount=50

#### Cuando hagamos el GET de:

GET http://localhost:8080/invoices

Veremos lo siguiente:



Ya que hemos enviado 2 peticiones POST. Si lo hiciéramos 4 veces y nos fuéramos al navegador observaríamos lo siguiente:

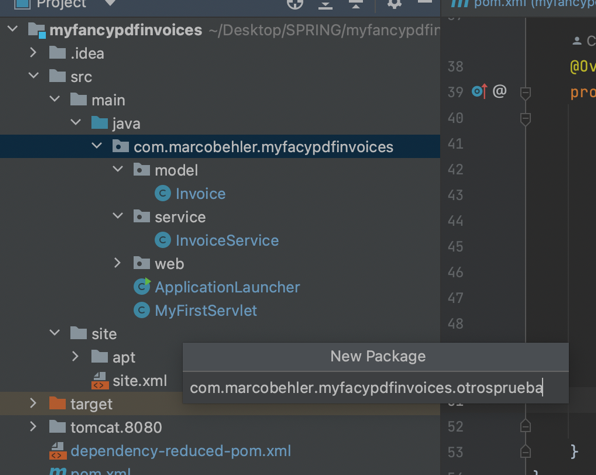
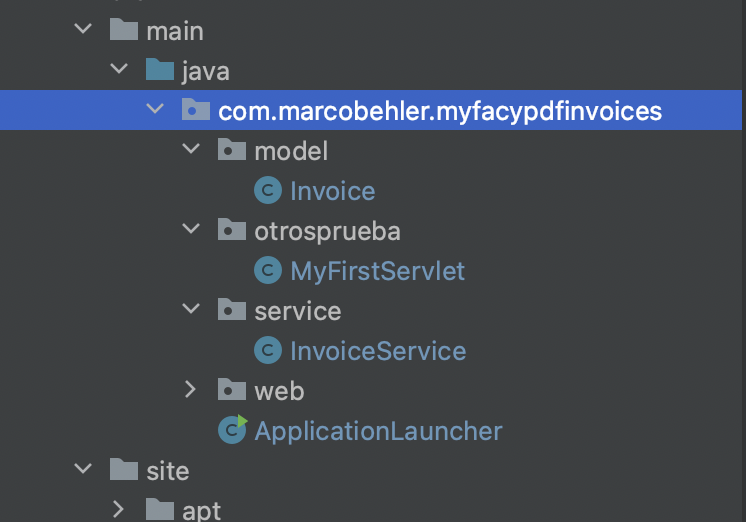


#### Refactorización de paquetes

Para ello vamos a poner:

* Las clases de dominio como (invoice) en un subpaquete al que llamaremos MODEL (model).
* Los servicios (InvoiceService), en un subpaquete llamado SERVICE (service).
* El servlet o aquello relacionado con classes web, en un subpaquete llamado WEB (web).
* Tenemos que crear el paquete raíz {groupId.appName}, en este caso: ***s***.
* El launcher o el main lo dejaremos dentro de este paquete raíz.

Quedando así:

****

**Creacion de una clase de aplicación global**

Ahora tenemos 2 variables en *MyFancyPdfInvoicesServlet,* una es ***invoiceService*** y la otra ***objectMapper***.

Esto está bien si solo tienes un HttpServlet y nadie más necesita acceder a estas dos clases.

Pero, ¿qué pasa si alguna otra clase también necesita convertir entre Java y JSON? Entonces también necesitan un ObjectMapper.

¿Y si algún otro módulo, como una herramienta CLI o un trabajo por lotes, necesita acceso a su *InvoiceService*?

Entonces, tener campos privados en la clase MyFancyPdfInvoicesServlet ya no será suficiente, ya que de manera óptima desearía un ritmo central donde pueda definir todos estos servicios y luego acceder a ellos de alguna manera.

Sin Spring, es muy probable que esto lleve a implementar el patrón de clase Application, la versión de un hombre pobre de la gestión de dependencias.

Para dar una solución crearemos una clase (Application) nueva que contenga estas 2 variables de forma estática (seguiremos el patron singleton) y lo haremos en un subpaquete al que llamaremos CONTEXT (context).

package com.marcobehler.myfacypdfinvoices.context;  
  
import com.fasterxml.jackson.databind.ObjectMapper;  
import com.marcobehler.myfacypdfinvoices.service.InvoiceService;  
  
public class Application {  
 public static final InvoiceService *invoiceService* = new InvoiceService();  
 public static final ObjectMapper *objectMapper* = new ObjectMapper();  
}

Una vez realizado esto, tenemos que refactorizar o limpiar un poco el servlet.

**Añadiendo una clase de usuario**

En la clase ***InvoiceService***, el método de crear un usuario genera facturas para ese usuario. Sin embargo, tenemos que asegurarnos de que el usuario realmente existe.

Entonces, agregaremos un objeto de dominio de usuario muy simple a su aplicación, con solo dos campos, id y nombre.

package com.marcobehler.myfacypdfinvoices.model;  
  
public class User {  
 private String id;  
 private String name;  
   
 // constructor por defecto  
 public User() {  
 }  
 // constructor de Usuario  
 public User(String id, String name) {  
 this.id = id;  
 this.name = name;  
 }  
  
 // setters y getters  
 public String getId() {  
 return id;  
 }  
  
 public void setId(String id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
}

**Añadiendo un servicio de usuario**

Solo con los usuarios no puede hacer mucho, también necesitamos un UserService correspondiente que le permita encontrar usuarios. Por ahora, va a crear una implementación ficticia de ese servicio, devolviendo siempre un usuario.

package com.marcobehler.myfacypdfinvoices.service;  
  
import com.marcobehler.myfacypdfinvoices.model.User;  
  
import java.util.UUID;  
  
public class UserService {  
  
 public User findById(String id) {  
 String randomName = UUID.*randomUUID*().toString();  
 // siempre "encuentra" al usuario, cada usuario tiene un nombre aleatorio  
 return new User(id, randomName);  
 }  
}

**Añadiendo una comprobación de validación**

En el InvoiceService haremos lo siguiente:

public Invoice create(String userId, Integer amount) {  
 User user = new UserService().findById(userId);  
 if (user == null) {  
 throw new IllegalStateException();  
 }  
   
 Invoice invoice = new Invoice(userId, amount, "http://www.africau.edu/images/default/sample.pdf");  
 invoices.add(invoice);  
 return invoice;  
}

Antes de crear una factura, ahora está construyendo un nuevo servicio de usuario y comprobando si esos usuarios existen.

User user = new UserService().findById(userId);  
if (user == null) {  
 throw new IllegalStateException();  
}

Si el usuario no existe, simplemente lanzaremos una excepción IllegalStateException().

**Refactorización de la clase Application**

Crear una nueva instancia de ***UserService*** dentro de ***InvoiceService*** no tiene demasiado sentido, ya que tiene esa clase de aplicación global que realiza un seguimiento de todas sus otras clases. Agregaremos un **UserService** dentro de la clase **Application**.

public class Application {  
 public static final InvoiceService *invoiceService* = new InvoiceService();  
 public static final ObjectMapper *objectMapper* = new ObjectMapper();  
 public static final UserService *userService* = new UserService();  
}

Por lo tanto, también cambiaremos la linea correspondiente en InvoiceService, para obtener el UserService de Application.

User user = Application.*userService*.findById(userId);  
if (user == null) {  
 throw new IllegalStateException();  
}

Pero aun así no es eficiente, debido a que ***InvoiceService*** aún necesita volver a llamar activamente a la aplicación para obtener una instancia de ***UserService***.

Para ello deberíamos realizar alguna inyección de dependencia.

**Primeros pasos para la inyección de dependencia**

En primer lugar, refactoricemos ***InvoiceService*** para que cualquiera que construya un nuevo ***InvoiceService*** también tenga que pasar un ***UserService*** válido.

package com.marcobehler.myfacypdfinvoices.service;  
  
import com.marcobehler.myfacypdfinvoices.model.Invoice;  
import com.marcobehler.myfacypdfinvoices.model.User;  
  
import java.util.List;  
import java.util.concurrent.CopyOnWriteArrayList;  
  
public class InvoiceService {  
  
 private final UserService userService;  
 private List<Invoice> invoices = new CopyOnWriteArrayList<>();  
  
 public InvoiceService(UserService userService) {  
 this.userService = userService;  
 }  
  
 public List<Invoice> findAll() {  
 return invoices;  
 }  
  
 public Invoice create(String userId, Integer amount) {  
 User user = userService.findById(userId);  
 if (user == null) {  
 throw new IllegalStateException();  
 }  
  
 Invoice invoice = new Invoice(userId, amount, "http://www.africau.edu/images/default/sample.pdf");  
 invoices.add(invoice);  
 return invoice;  
 }  
  
}

Ahora arreglaremos la clase ***Application***, ya que ya no se compilará más.

public class Application {  
 public static final UserService *userService* = new UserService();  
 public static final InvoiceService *invoiceService* = new InvoiceService(*userService*);  
 public static final ObjectMapper *objectMapper* = new ObjectMapper();  
}

***InvoiceService*** ya no tiene que preocuparse por la fontanería, pero se le puede garantizar que se le *inyectará* un ***UserService*** que funcione