Primera edición

Septiembre 2022

Spring Boot

Arquitectura de Back End

Diseño de un microservicio profesional

Rest API | Java 17 | Lombok | JPA | Docker | PostgreSQL | Postman

Útimas versiones de JAVA y Spring

Septiembre 2022

Rafael Benedettelli


Primera edición
Septiembre 2022
Spring Boot
Arquitectura de Back End
Diseño de un microservicio profesional
Rest API | Java 17 | Lombok | JPA | Docker | PostgreSQL | Postman
Últimas versiones de JAVA y Spring
Septiembre 2022
Rafael Benedettelli

Spring Boot

Arquitectura de Back End

Contenidos

Autor

Rafael Benedettelli es Ingeniero en informática por la Universidad de Palermo, Buenos Aires, República Argentina y certificado Java por Oracle.

Desde el año 2002 ha liderado proyectos de software para Argentina, Uruguay, Ecuador, Luxemburgo y España. Se ha desenvuelto como arquitecto de software en proyectos para el sector bancario, farmacéutico y de ingeniería civil.

Actualmente dirige su consultora Ruby Canadian Software radicada en British Columbia, Canadá. En lo académico imparte capacitaciones y forma equipos de desarrollo para empresas de Argentina y España.

Agradecimientos

Quisiera dedicar esta obra a mis padres Carlos Alberto Enrique Benedettelli y Marta Lakovich que ya no están aquí. Gracias por haberme enseñado los valores más importantes de la vida.

Capítulo 1

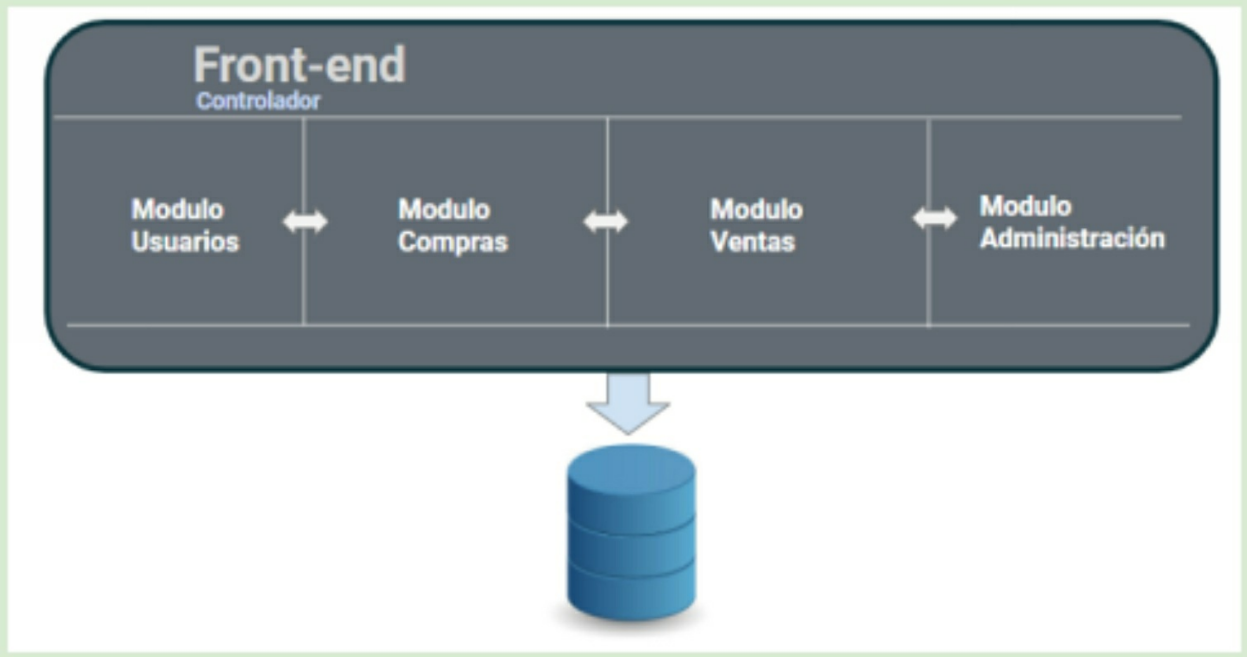
1.1 Introducción

Introducción a las arquitecturas actuales de sistemas

1.1 Introducción a la arquitectura de microservicios

A mediados de la década del 2000, las arquitecturas de sistemas tuvieron que verse obligadas a transformarse para poder atender a las nuevas demandas del entorno.

El surgimiento de nuevos dispositivos, sobre todo los móviles como teléfonos inteligentes, tablets, Smart TV, consolas de videojuegos por mencionar algunos, obligaron a cambiar el paradigma de ciertos estándares de la época. Estos cambios significativos implicaron tener que abandonar de a poco la idea de un sistema concentrado, conocido como el famoso *"monolito"*, que reúne toda la lógica de negocio y presentación en una sola pieza de software.

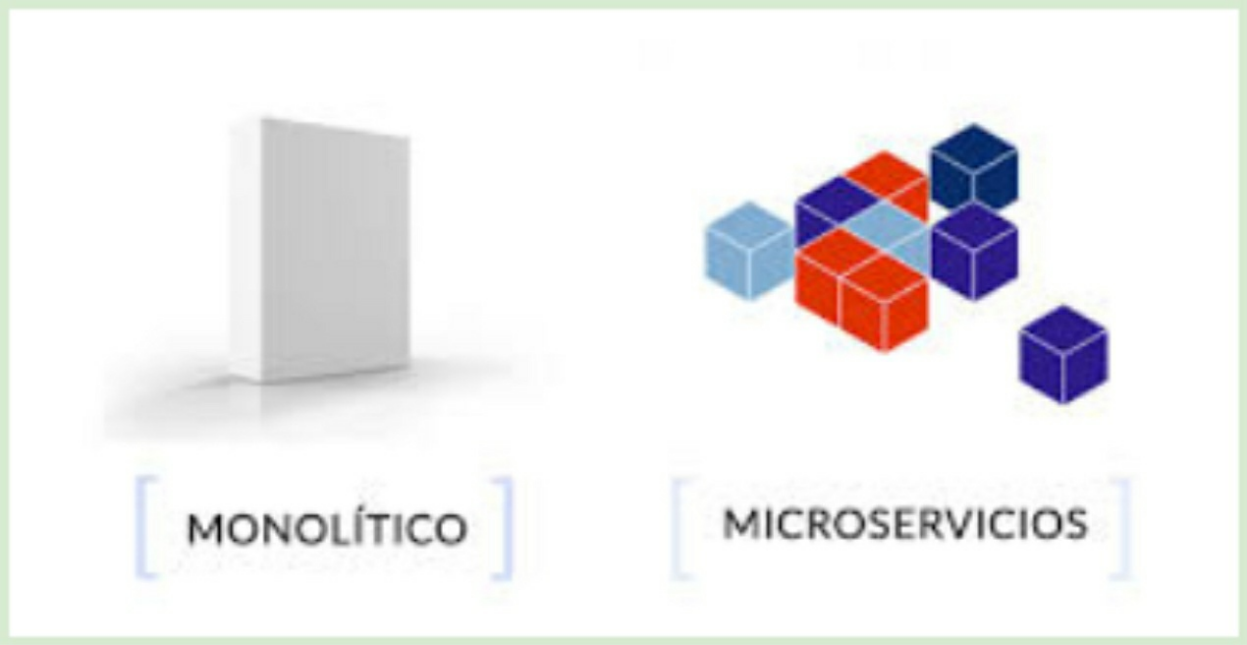


Radiografía de un "monolito". Una pieza gigante de software que incluye 3 capas (presentación, controlador y modelo) y mantiene una división lógica interna.

Es en ese entonces que comenzamos a escuchar de esta nueva separación mencionada incluso en las bolsas de trabajo como *"Back End"* y *"Front End"*. El primer paso era, sin dudas, dividir en dos piezas grandes estas dos capas.

Y si de dividir se trata, también comenzó a surgir la idea de dividir en el Back End al gigante monolito en varios subsistemas que en su conjunto brindan una solución para un dominio específico.

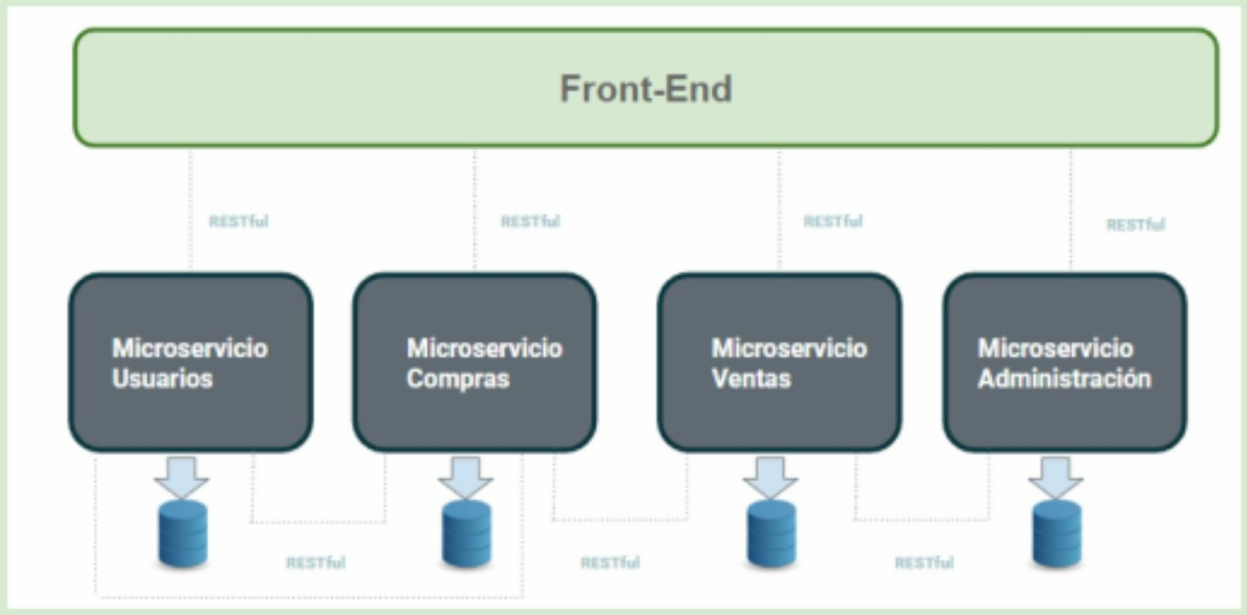
Esta división de un sistema en varios que puedan compartir información mediante protocolos ligeros de comunicación como REST comenzó a conocerse como arquitectura de microservicios.



Esta nueva división permite varios beneficios tanto en las fases de desarrollo y producción. Uno de los enfoques más comunes para llevar a cabo esta división es por “funcionalidad”. Es decir, imaginemos una plataforma de ventas online. Esta la podríamos dividir en varios microservicios de acuerdo a los subdominios más relevantes como compras, ventas, stock, usuarios, administración, promociones, etc.

1.2 Beneficios de la arquitectura de microservicios

Esta orientación, en la fase de desarrollo, permite organizar equipos de manera más ordenada y focalizada, ya que podremos contar con un programador por cada microservicio que se enfoque exclusivamente en esta mini solución que conforma luego una parte de todo el sistema. Incluso, esta división, permite también que algunos microservicios puedan ser implementados en otros lenguajes, claro está, siempre y cuando la comunicación entre el "mundo exterior" se encuentre estandarizada en un mecanismo agnóstico. Como por ejemplo REST que es el más utilizado pero también cabe la posibilidad de utilizar otros protocolos incluso binarios.



Misma solución pero orientada a microservicios. En este caso el monolito se divide en distintos subsistemas que ahora comparten información mediante REST. Como se puede observar, el Front End es un artefacto separado que consume datos mediantes las interfaces externas de los microservicios. Este esquema propone una base de datos por cada microservicio. Existen distintas tipologías para la organización de microservicios. Generalmente existe un micro de orden jerárquico mayor que hace el rol de "Gateway" para atender solicitudes de frontales u otros sistemas corporativos.

En materia de desarrollo, obtenemos también ventajas significativas en mantenimiento y escalabilidad, ya que un cambio o arreglo efectuado en un microservicio no impactará en otro. Por supuesto, siempre y cuando lo que no se altere sea la interfaz externa de comunicación del artefacto.

Tal vez en materia de despliegue podemos encontrar cierta complejidad, ya que ahora, en lugar de lidiar con solo un artefacto, estamos lidiando con muchos. En este sentido la coordinación y los procesos de DevOps deben ser bien diseñados con el propósito de lograr un despliegue lo más coordinado posible.

No obstante, uno de los mayores provechos de esta nueva arquitectura la encontraremos

en producción. Al contar con varios subsistemas gozaremos de la libertad de poder realizar despliegues en infraestructuras de hardware independientes. Hoy en día, por ejemplo, una de las opciones más utilizadas son los contenedores Docker.

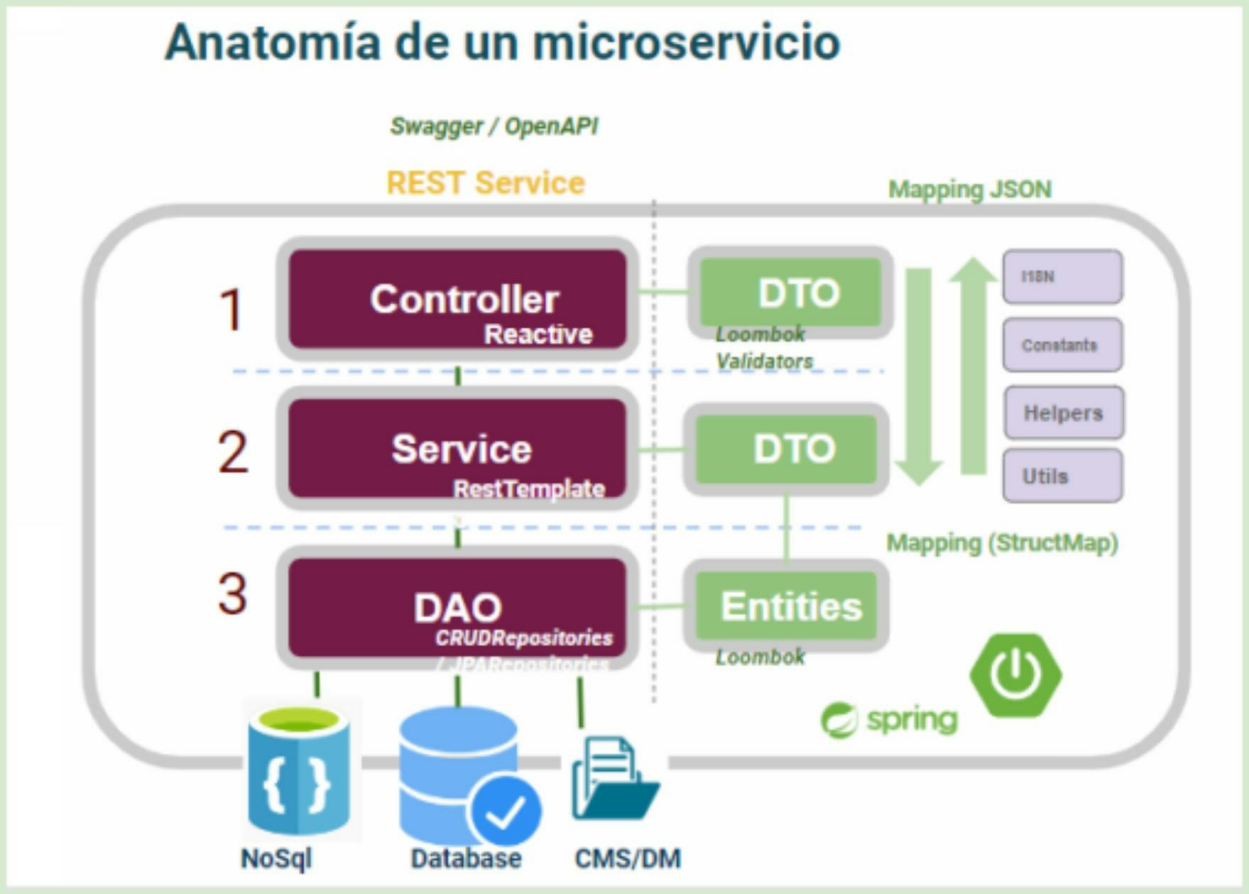
Imaginemos que el microservicio de ventas de nuestra aplicación es el más concurrido por los usuarios. De acuerdo a esta demanda podremos establecer más recursos computacionales para este micro en concreto y no tantos para el micro de stock por ejemplo.

Supongamos que el micro de stock lo usan solo cuatro usuarios administradores, mientras que el micro de ventas es concurrido por más de diez mil usuarios diariamente. En este caso, nuestra prioridad será siempre el microservicio de ventas. No solo por la cantidad de usuarios sino también porque esos usuarios son los clientes, y por supuesto, son los que ingresan dinero en nuestra empresa.

De acuerdo a este escenario, ante cualquier eventual caída en el microservicio de stock, esta, no alteraría en absoluto el normal funcionamiento del microservicio de venta o cualquier otro. En este caso nos focalizamos exclusivamente en solucionar el problema del servicio de stock mientras que nuestra solución sigue sirviendo al cliente final sin problemas.

1.3 Anatomía interna de un microservicio

Durante este libro desarrollaremos un microservicio completo y totalmente profesional delineado por un diseño en capas y una arquitectura interna lo más flexible y modular posible.



Como se puede observar en la imagen anterior, y en color violeta se identifican las tres capas de la estructura interna. No obstante, en color verde se identifican aquellos componentes que transitan mediante las tres capas de nuestra aplicación.

Para la implementación de nuestro microservicio basado en Java Spring Boot, nos basaremos en esta radiografía y desarrollaremos cada una de las capas y componentes necesarios para alinearnos con este diseño.

Capítulo 2

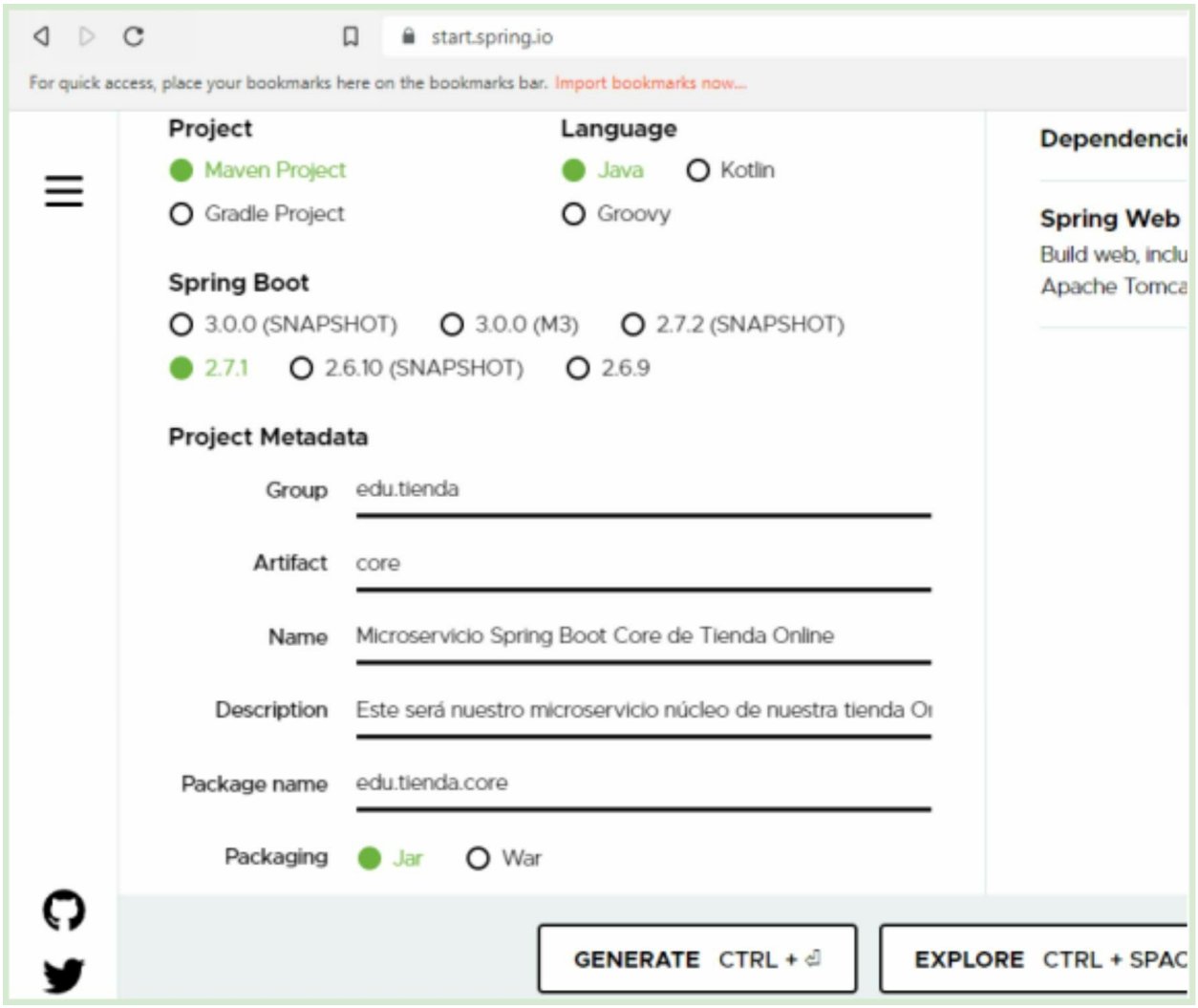
2.1 Iniciar un proyecto Spring Boot

Sin perder tiempo, vamos a la práctica

2.1.1 Crear un proyecto Spring Boot

En este apartado, vamos a crear un proyecto Spring Boot desde cero. Para eso, vamos a utilizar el asistente oficial de Spring: Spring Initializr.

Accedemos a la página https://start.spring.io y se nos presentará la siguiente pantalla:



Este intuitivo y fácil asistente nos permitirá generar un proyecto limpio desde cero en Spring Boot. Solamente tenemos que configurar unos simples parámetros. En primer lugar, mantenemos los valores por defecto para Project, Language y Spring Boot como figura en la imagen superior. A continuación se listaran los parámetros y los valores que debemos configurar:

Project: Maven Project

Esto indica que será un proyecto Java de tipo Maven. Es decir, que será un proyecto java administrado por el Project Management Maven, un poderoso gestor de proyectos y dependencias.

Language: Java

Esto indica que será un proyecto Spring Boot donde el lenguaje de programación principal será Java.

Spring Boot: 2.7.0

Al momento de escribir este libro, Spring Boot 2.7.0 es la última versión estable del framework. Ahora vamos a darle una carátula al proyecto. Se trata de asignar al artefacto los datos de título que maven necesita para identificarlo. Estos son solo descriptores técnicos y tendrán visibilidad únicamente para el equipo de desarrollo.

Para nuestro proyecto de prueba y con fines educativos vamos utilizar los siguientes descriptores tal como se presentan a continuación:

Project Metadata:

Group: edu.tienda

Se refiere al grupo de artefactos al cual pertenece este proyecto. Generalmente se escribe como “Domain Reverse Name”, es decir, como un nombre de dominio pero de forma reversa donde “edu” hace referencia a que se trata de una solución de tipo educativa. Generalmente en los ambientes de trabajo y producción se suele utilizar la nomenclatura “*com.nombreEmpresa*”.

Artifact: core

Se refiere al nombre del proyecto en particular, que en este caso, lo identificamos con el titulo “*core”*, ya que se trata de un microservicio Spring Boot que oficiara con el role de ser el central o más importante de varios microservicios que formarán parte de una solución.

Name: Microservicio Spring Boot Core de Tienda Online.

Aquí agregaremos un nombre más coloquial y descriptivo del proyecto.

Description: Este será nuestro microservicio núcleo de nuestra tienda Online.

Se trata de una descripción más larga y detallada.

Package Name: edu.tienda.core

Cuando se genere el “scaffold” del proyecto (Estructura de carpetas del proyecto) se generará automáticamente un paquete siguiendo estos nombres.

Packaging: JAR

El asistente nos permite dos tipos de empaquetamientos. JAR es por defecto el más utilizado, ya que nos permite ejecutar la aplicación de manera independiente y autocontenida.

Java: 17

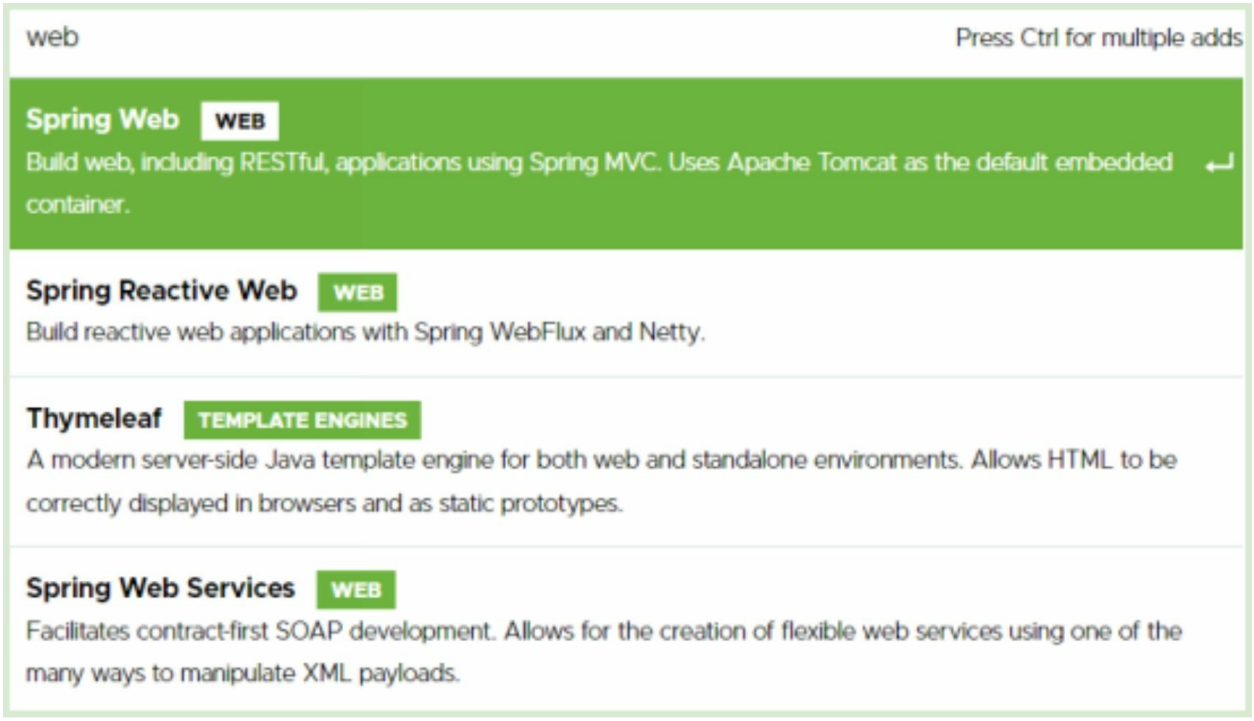
Este será el parámetro más sensible de todos. Si bien Spring va a sugerir por defecto la versión más reciente de java, recomendamos echar un ojo a la instalación java de nuestro equipo.

Importante: Todos estos parámetros pueden ser cambiados una vez generado el proyecto Spring sin ningún tipo de inconvenientes, ya que van a residir en el archivo “pom.xml” situado en la ruta base de la estructura de carpetas.

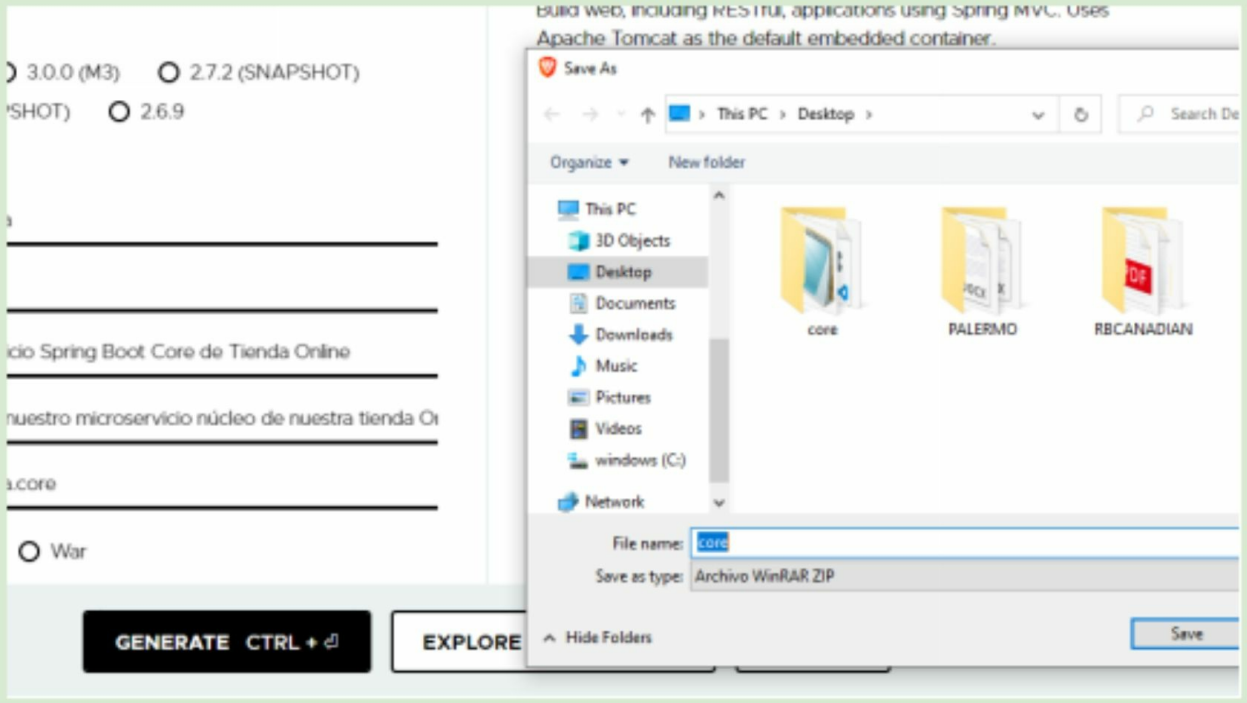
Por último, solo nos queda agregar una librería fundamental con la cual trabajaremos:

Dependencies: Spring Web

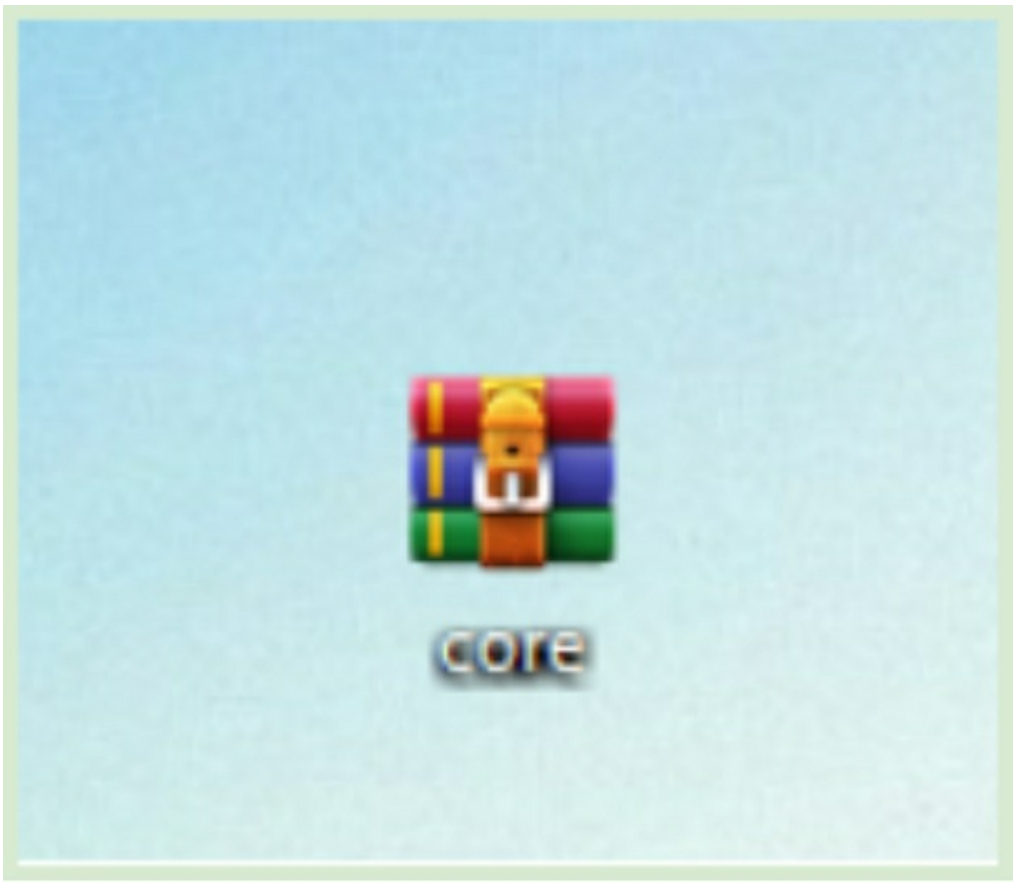
Hacer click en el botón “add dependencies” y buscar la librería “Spring Web”. Cuando aparezca en el listado de sugerencias, darle un click para agregarla a nuestra solución.



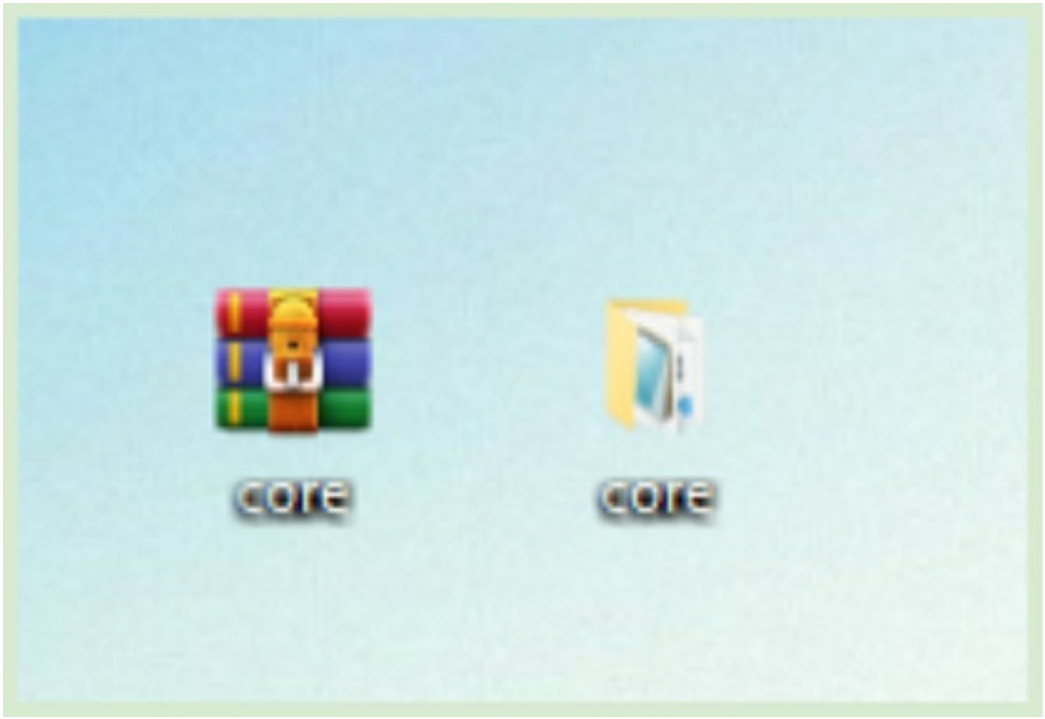
¡Genial!, ya tenemos todo listo para generar nuestro primer proyecto Java Spring Boot. Solo resta hacer click en el botón inferior ”GENERATE”.



Al hacer click en “GENERATE”, nos permitirá descargarnos el proyecto en nuestro disco. Seleccionar una carpeta que nos quede cómoda para guardarla. En nuestro caso lo guardaremos en el escritorio de nuestro ordenador. A continuación, se descargara el nuevo proyecto Spring comprimido en nuestro escritorio.



Deberán extraer todo el contenido del archivo comprimido en la misma ubicación tal que nos quede una carpeta como se puede apreciar en la siguiente imagen.



En la siguiente sección abriremos nuestro proyecto con la herramienta IntelliJ y lo ejecutaremos.

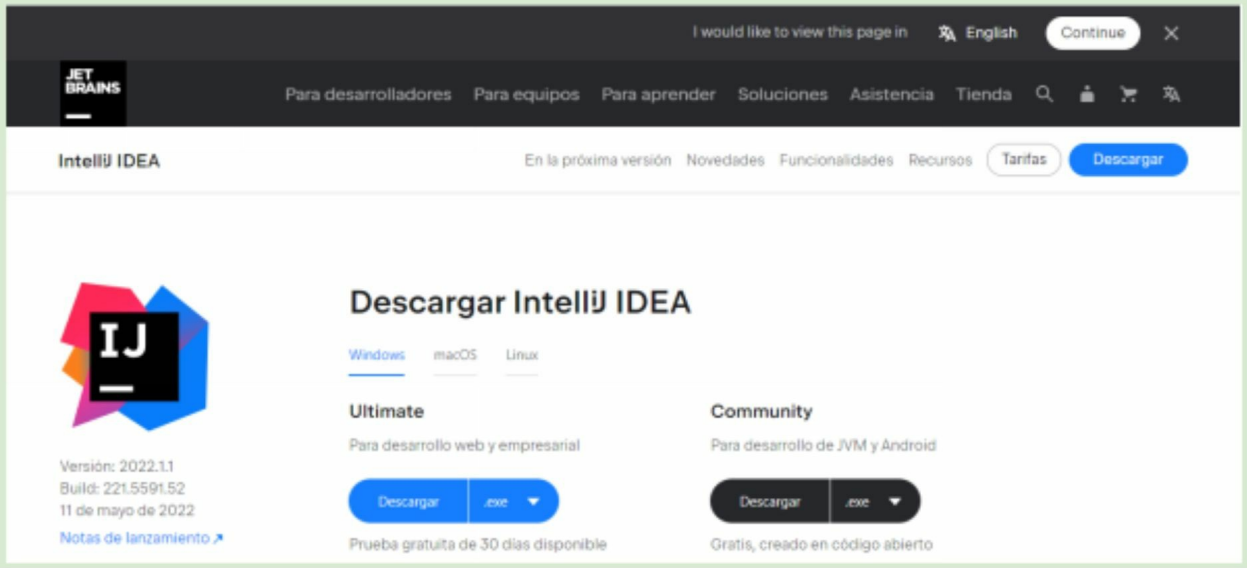
2.2 Ejecutando Spring Boot

Vamos a ejecutar nuestro primer proyecto Spring Boot.

2.2.1 Importar el proyecto generado con IntelliJ

Durante todo este libro vamos a trabajar con la herramienta IntelliJ. Un muy sofisticado y práctico IDE Java que nos permitirá trabajar con nuestra solución Spring Boot de manera cómoda y agradable.

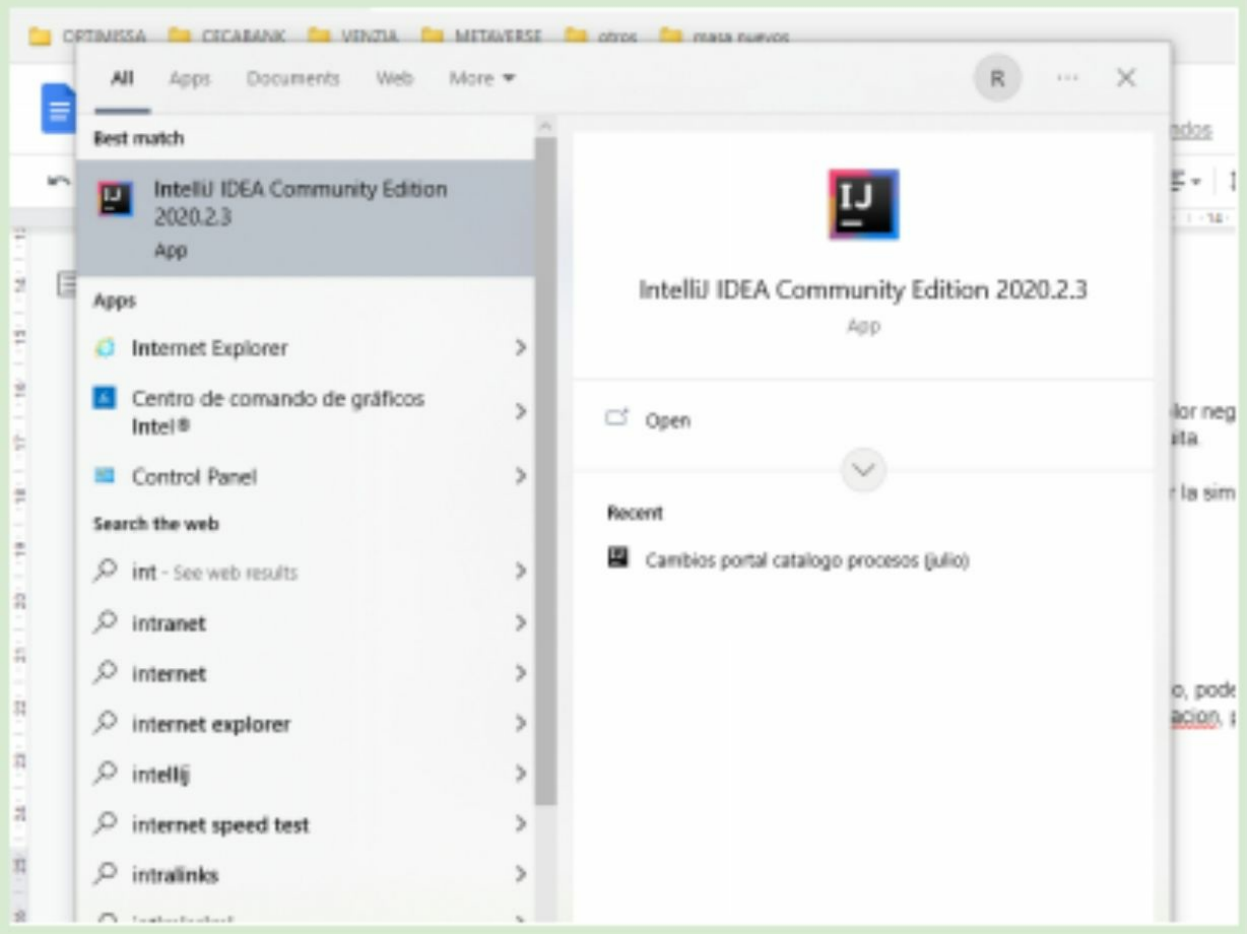
Podemos descargar el IDE IntelliJ de la siguiente página https://www.jetbrains.com/es-es/idea/download/#section=windows y se nos presentará esta pantalla:



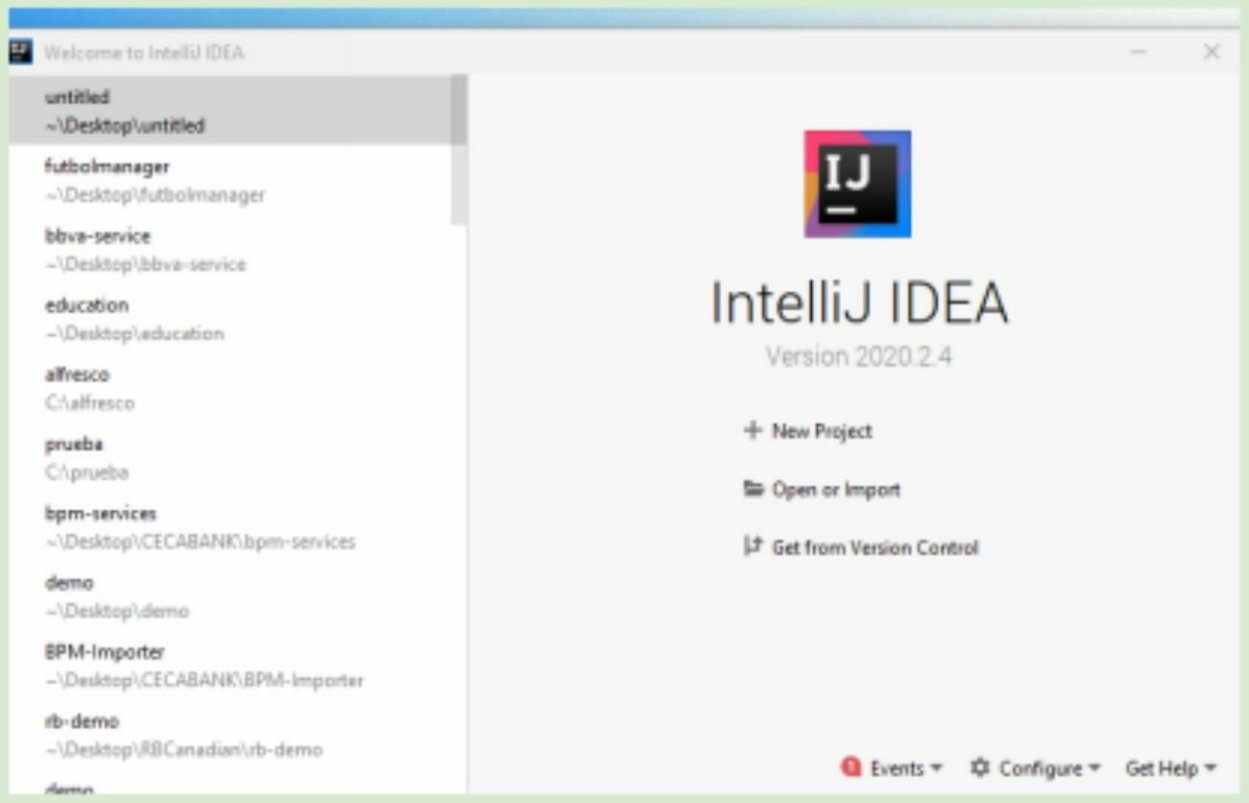
Una vez situados en esta página, hacemos click en el botón “descargar” de color negro. Nos estaremos descargando la versión community de este software, que es gratuita. Una vez descargado el archivo “.exe”, simplemente lo ejecutamos para realizar la simple instalación que no será más que un “paso a paso”.

Ejecutando IntelliJ

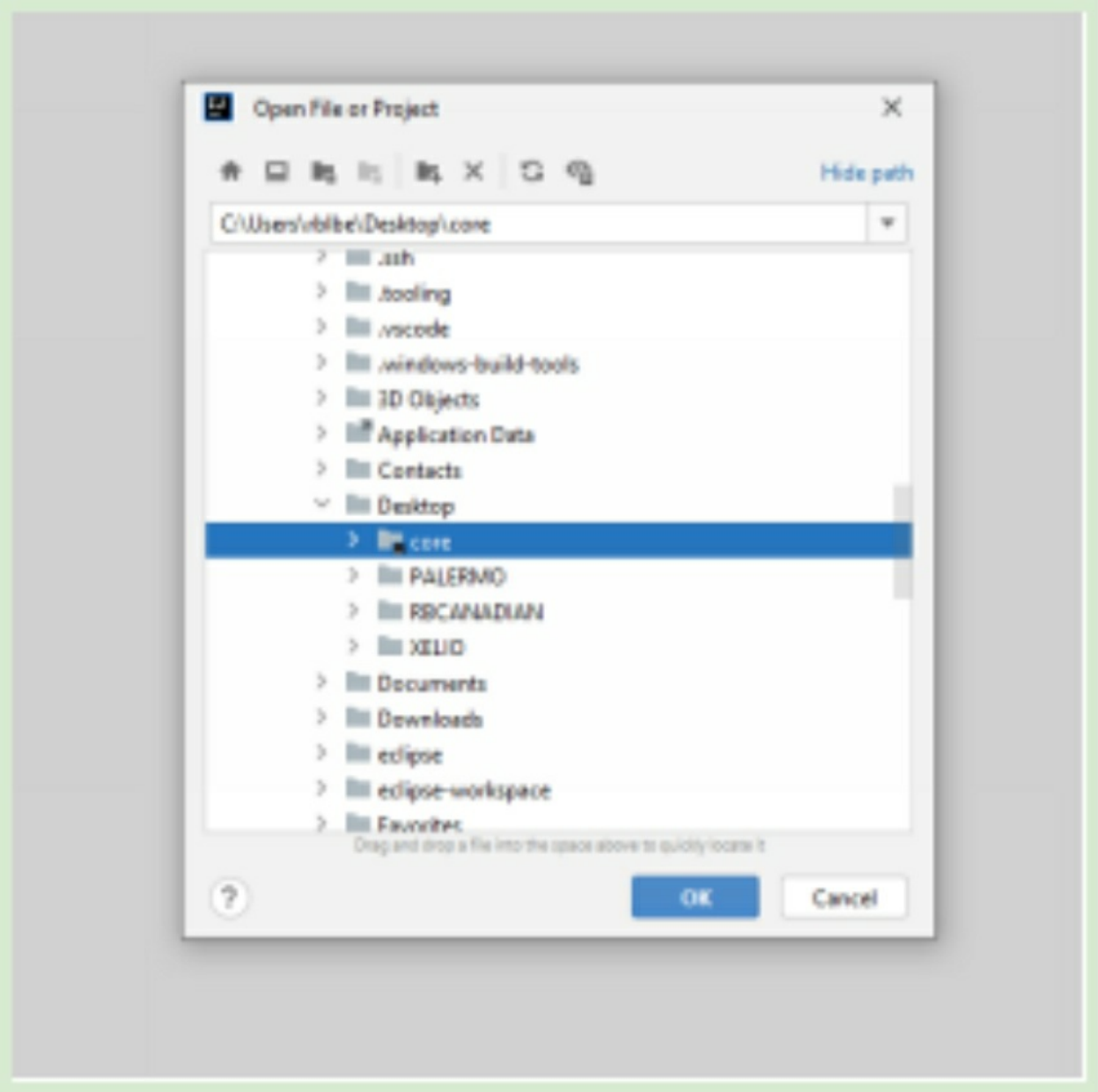
En el cuadro de diálogo inferior de windows o haciendo click en el botón “inicio”, podemos encontrar el icono de IntelliJ y ejecutarlo. También, dependiendo de la instalación, puede que dispongamos de un acceso directo en el escritorio.



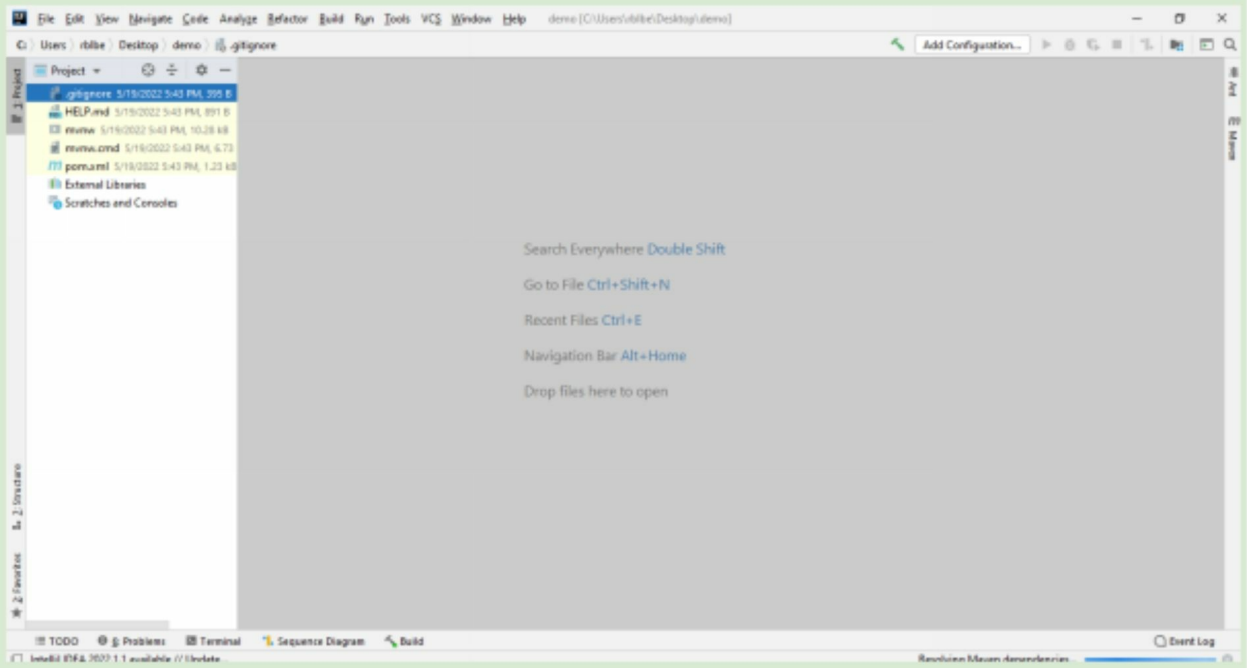
Al ejecutar IntelliJ, se nos abrirá un pequeño asistente como se muestra en la imagen:



Desde aquí, se podrá crear un proyecto nuevo desde cero o importar uno existente. Para nuestro caso vamos a importar el proyecto que hemos generado con “Spring Initializr”. Hacemos click en “Open or Import” y se nos abrirá un asistente para encontrar la carpeta de nuestro proyecto.



Localizamos la carpeta con el nombre “core” y hacemos click en “ok”. Se abrirá IntelliJ con la intención de cargar el proyecto importado.

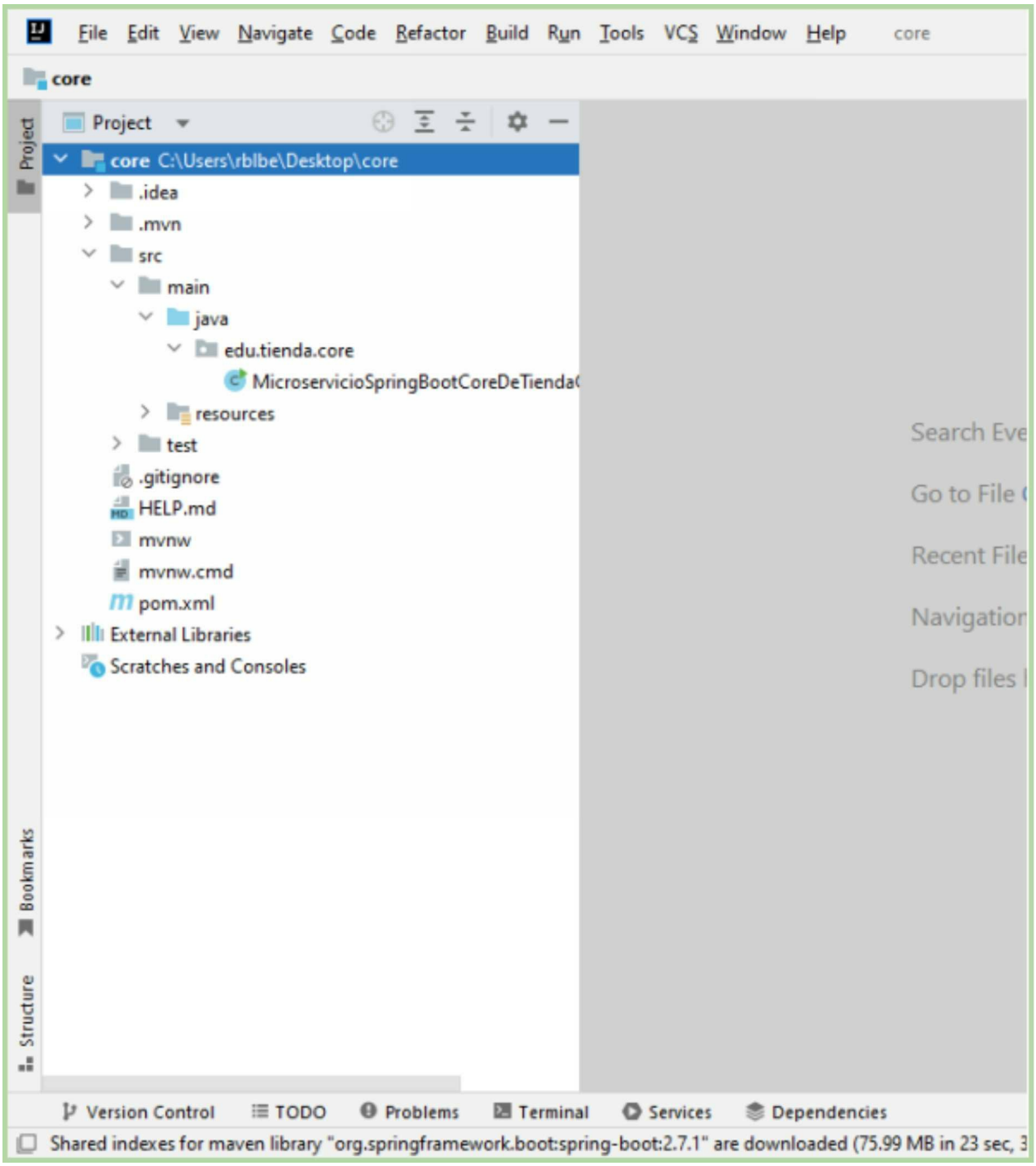


Deberemos esperar un momento hasta que maven resuelva las dependencias. Esto puede tomar unos minutos.

En el ángulo inferior derecho de nuestra pantalla podremos ver como maven descarga dependencias y plugins necesarios para la construcción. Deberemos esperar a que la barra de progreso azul finalice su descarga correspondiente.

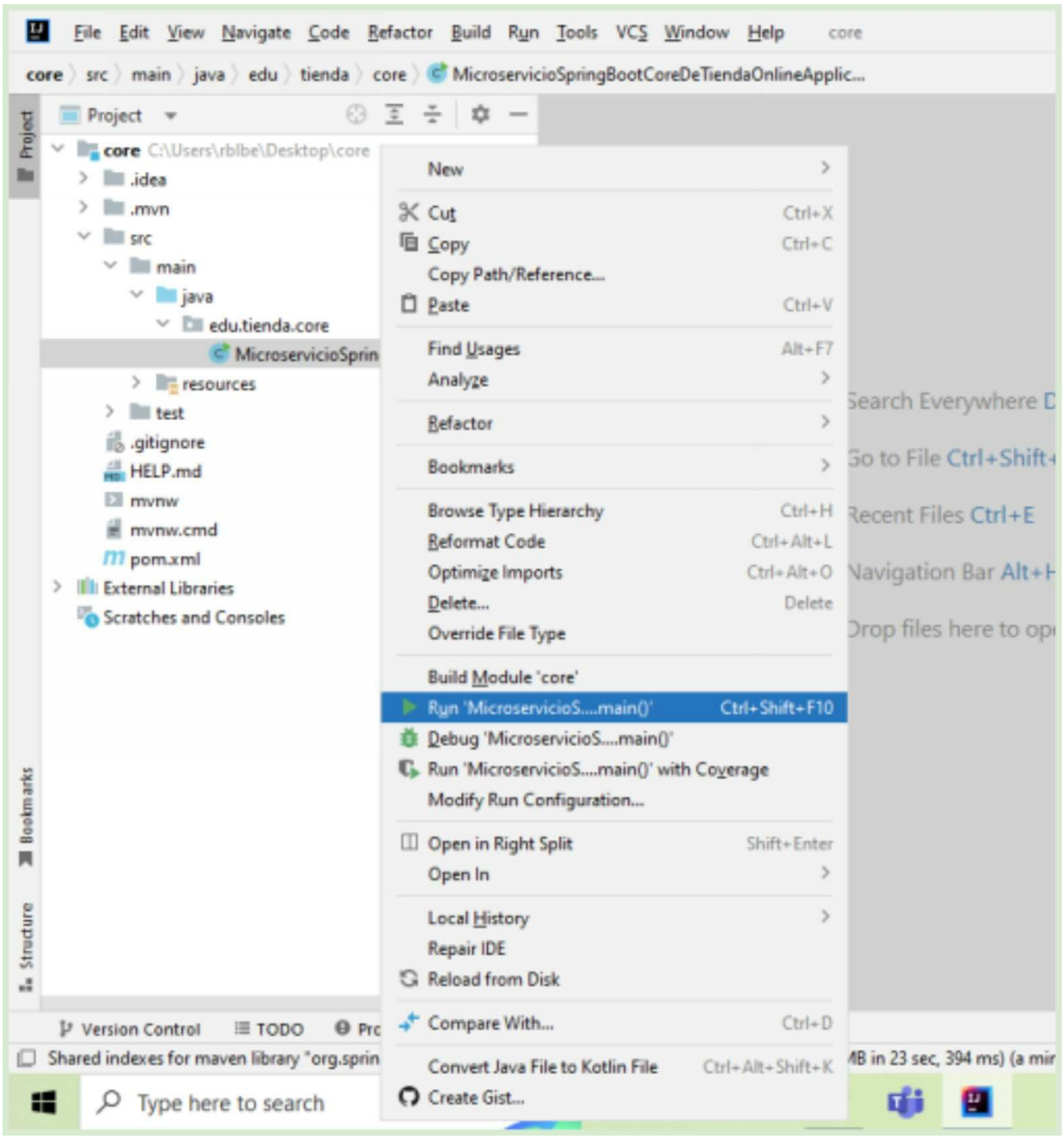
2.2.2 Estructura del proyecto

Una vez finalizada la descarga de dependencias, podemos echar un vistazo a la estructura básica de carpetas que conforman el proyecto. En la jerga, esta estructura de carpetas se la suele conocer como “Scaffold”.


File Edit View Navigate Code Refactor Build Run Tools VCS Window Help core
core
Project
core C:\Users\rblbe\Desktop\core
.idea
.mvn
src
main
java
edu.tienda.core
MicroservicioSpringBootCoreDeTienda
resources
test
.gitignore
HELP.md
mvnw
mvnw.cmd
pom.xml
External Libraries
Scratches and Consoles
Search Eve
Go to File
Recent File
Navigatior
Drop files
Bookmarks
Structure
Version Control
TODO
Problems
Terminal
Services
Dependencies
Shared indexes for maven library "org.springframework.boot:spring-boot:2.7.1" are downloaded (75.99 MB in 23 sec, 3

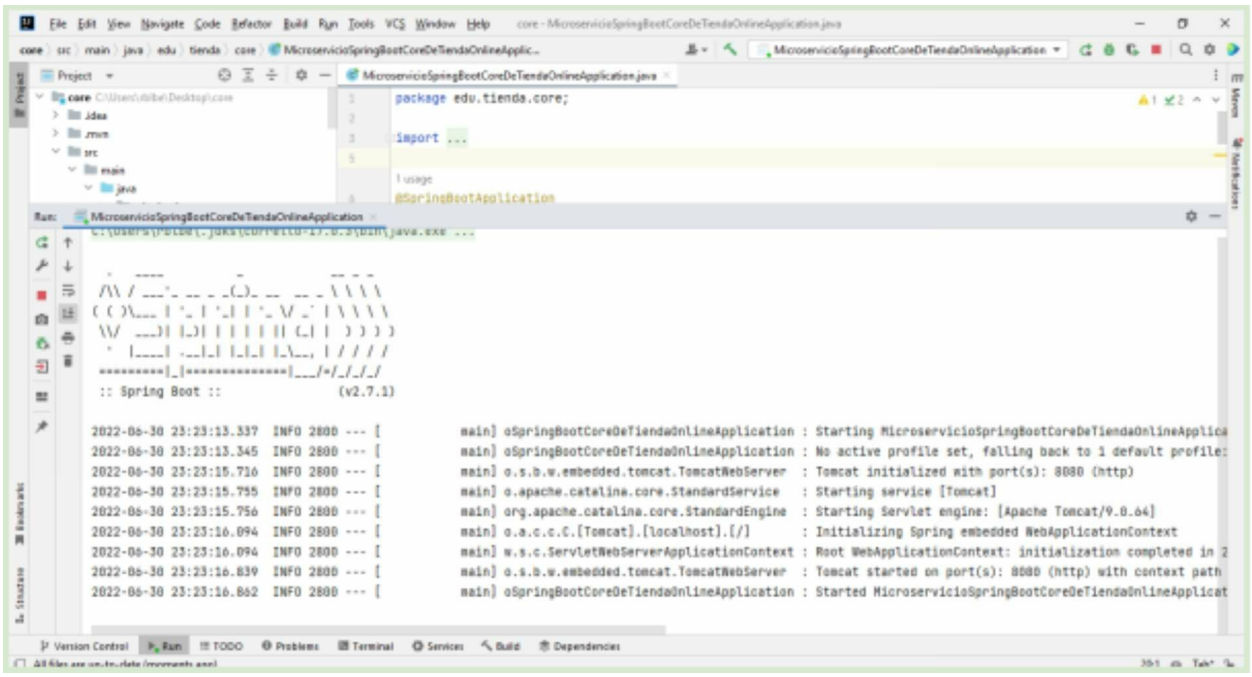
Ejecutar Spring Boot

Vamos a ejecutar el proyecto Spring Boot. Para eso, iremos abriendo una serie de carpetas a partir de la carpeta con nombre “src”. Nos situaremos en la ubicación “src/main/java”. En esta posición, abriremos el paquete *“edu.tienda.core”*. Una vez ubicados aquí, solo resta ejecutar el archivo Java visible con nombre *“MicroservicioSpringBootCoreDeTiendaOnlineApplication”*.



Hacemos click con el botón derecho del mouse sobre el archivo *"MicroservicioSpringBootCoreDeTiendaOnlineApplication"* seguido de un clicik en la opción *"Run MicroservicioSpringBootCoreDeTiendaOnlineApplication.main()"* que presenta el menú flotante.

Spring boot iniciará la ejecución realizando el "bootstrap" necesario de todos sus componentes. Una vez ejecutado el "startup", se podrá apreciar en la consola del IntelliJ el logo de Spring y los siguientes mensajes de log como se muestra a continuación:



Las dos últimas trazas de log son las que nos interesan por el momento, ya que la ante última nos indica que Spring se ha ejecutado como servicio escuchando en el puerto 8080 mediante el protocolo HTTP.

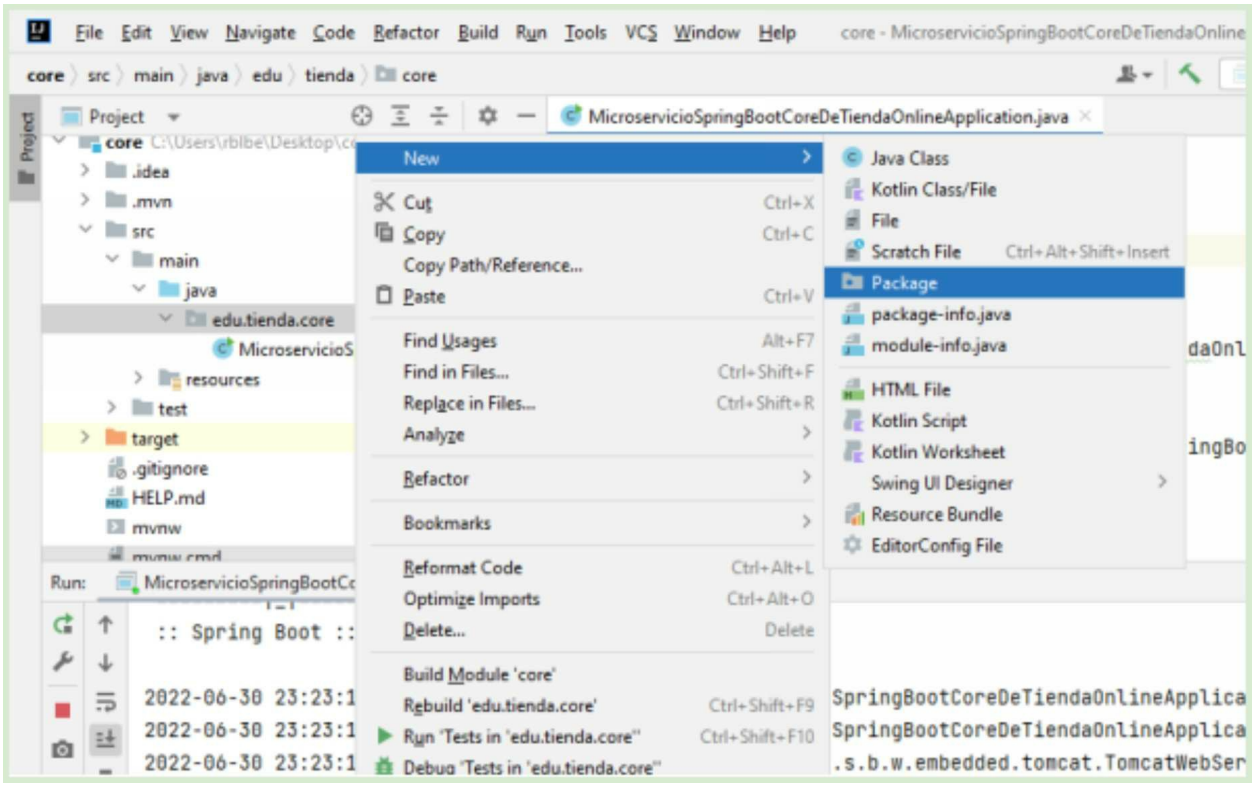
La última traza, es la que indica que el servicio Spring ha podido arrancar sin problemas. En este punto, Spring se está ejecutando como servicio suscrito al puerto 8080 y listo para recibir cualquier tipo de peticiones. En la siguiente sección comenzaremos a programar nuestras primeras líneas de código para ejecutar el famoso “Hola Mundo”.2.3 “Hola Mundo” Spring Boot

Crearemos nuestra primer clase en Spring Boot

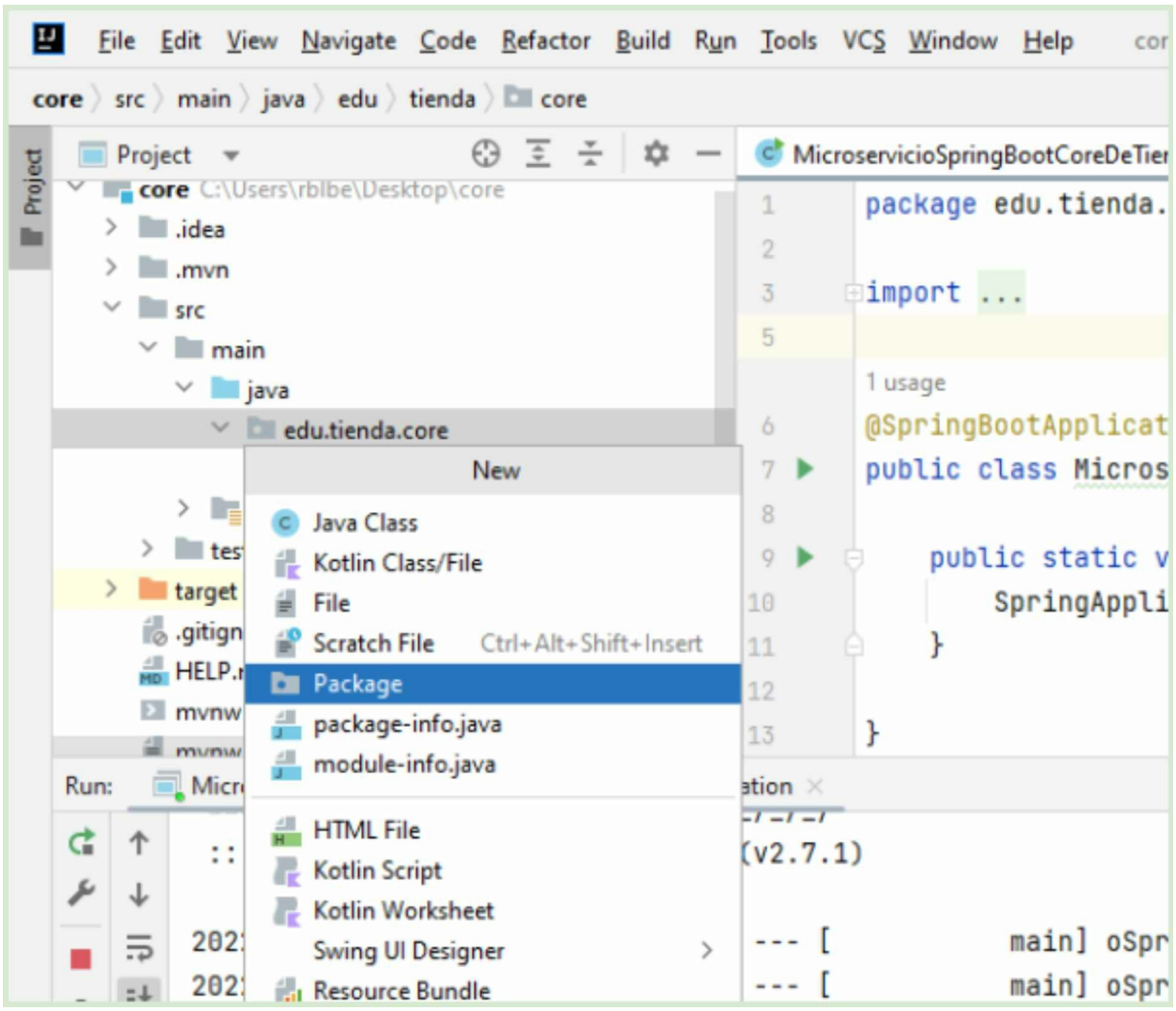
2.3.1 “Hola Mundo” en Spring Boot

Ahora realizaremos la famosa ceremonia de bautismo que es habitual en cualquier lenguaje o tecnología nueva.

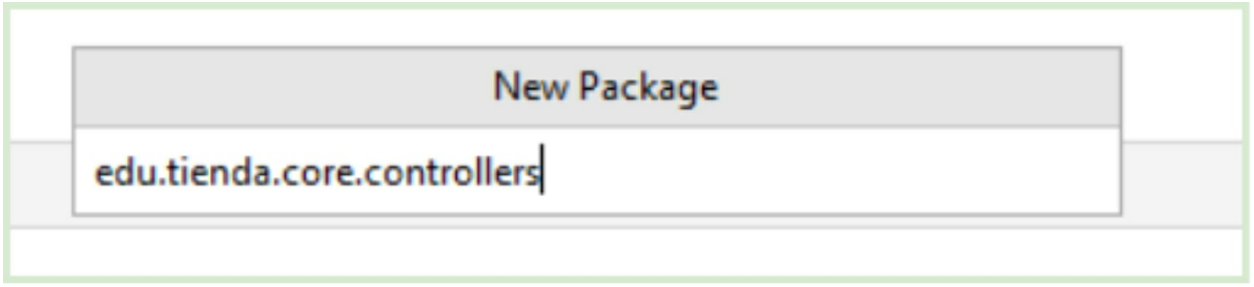
Nos situaremos en el paquete *“edu.tienda.core”* y haciendo click con el botón derecho del mouse se nos presentará un menú emergente que nos permitirá crear un nuevo sub-paquete.



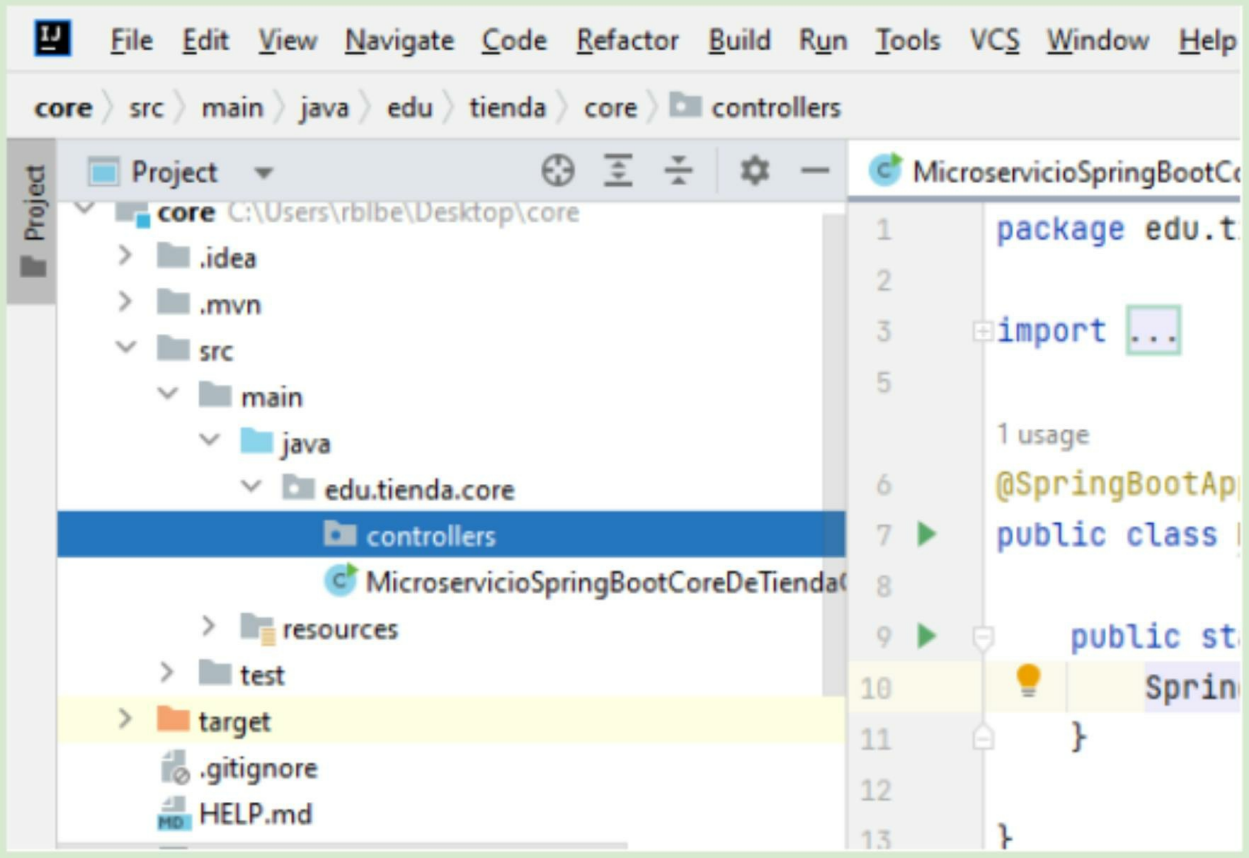
Opcionalmente, si nos situamos en el paquete *"edu.tienda.core"* podremos crear un paquete java de manera rápida presionando la combinación de teclas "ALT + INS". Esto nos presentará un menú rápido para la creación de componentes.



A continuación, hacemos click en “package” y se nos presentará un asistente para asignar un nombre al paquete.

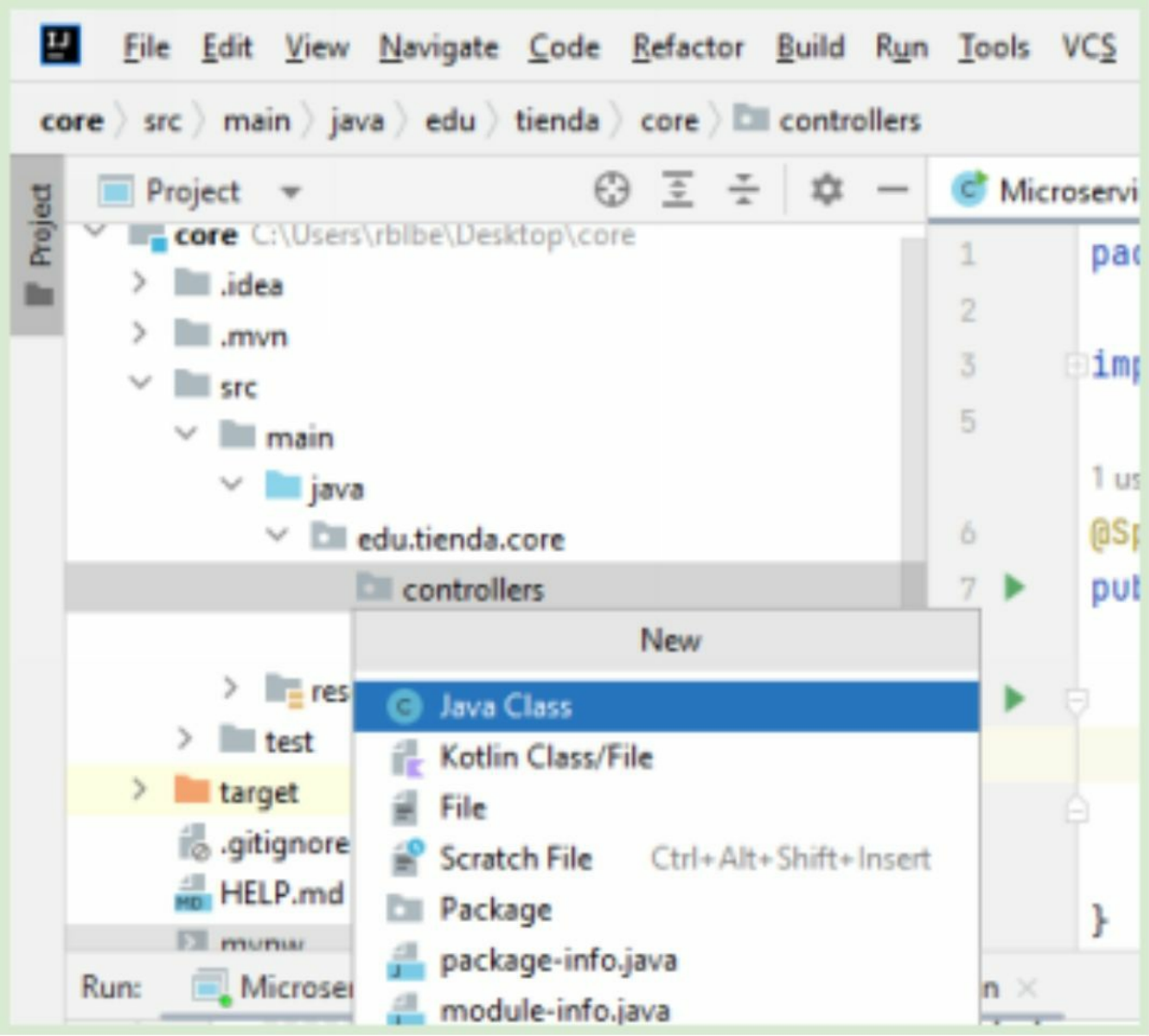


Escribimos el nombre *“controllers”* y obtendremos la siguiente estructura:

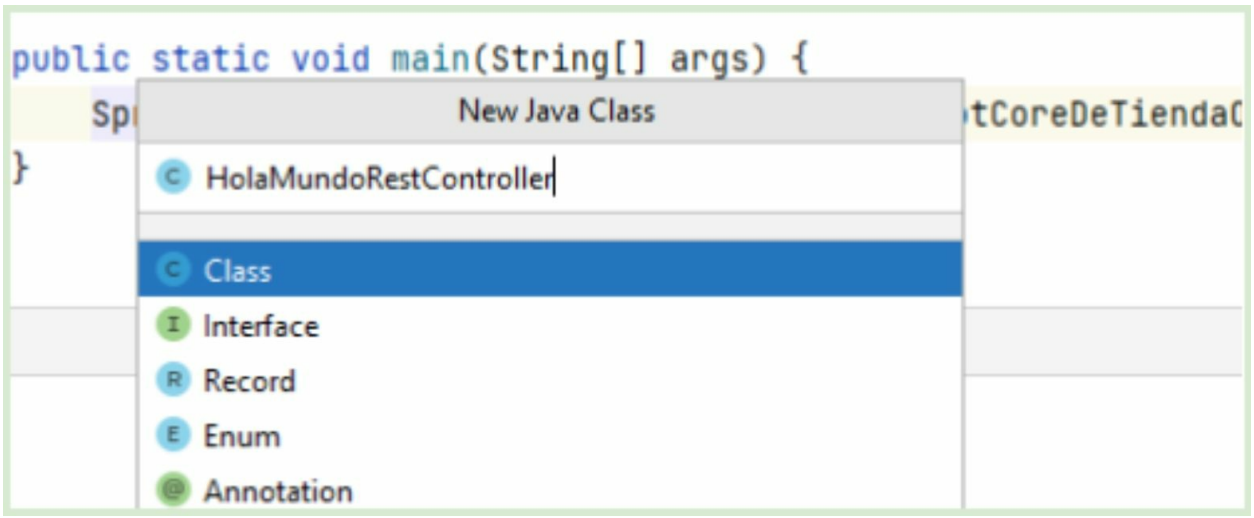


Como pueden apreciar, se ha creado el paquete “controllers” dentro del paquete *“edu.tienda.core”*.

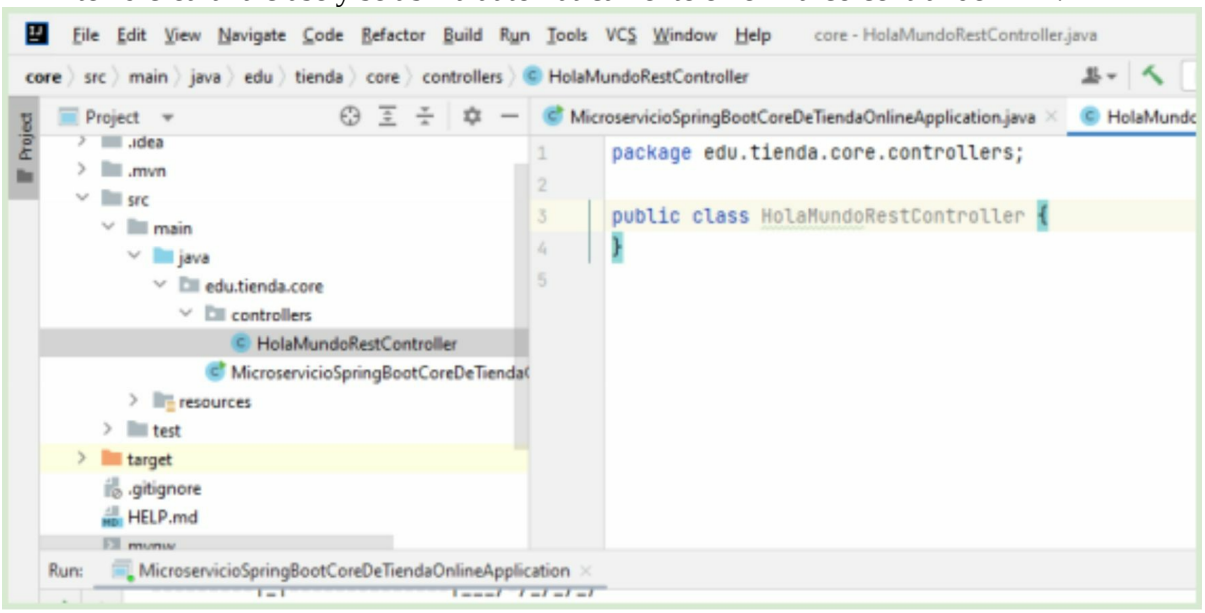
Ahora crearemos una clase java dentro de este paquete con el nombre *“HolaMundoRestController”*. Para eso, volvemos a presionar “ALT+INS” situados en el paquete *“controllers”*. Se volverá a presentar el menú de creación rápido y haremos click en “Java Class”.



En el siguiente paso se presentará un input para nombrar nuestra clase. Escribimos "*HolaMundoRestController*" y la tecla "Enter".



IntelliJ creará la clase y se abrirá automáticamente en el marco central del IDE.



.

Definiremos un simple método java que servirá para retornar un saludo cuando sea ejecutado:

```java
package edu.tienda.core.controllers;

public class HolaMundoRestController {

    public String saludo(){
        return "Hola Mundo Spring Boot";
    }


}
```

Como se puede apreciar, se trata de una simple y ordinaria clase java que contiene un simple y ordinario método. Hasta aquí nada nuevo.

Es en este punto es en el que comenzaremos a “decorar” nuestra clase java con las anotaciones de Spring, la cual la dotaran de la potencia necesaria para transformar esta clase común y silvestre en un servicio web de tipo REST que estará expuesto y listo para ser consumido desde cualquier explorador.

Para lograr este objetivo, vamos a decorar, en primer lugar, la clase con la anotación @RestController como se muestra en la siguiente figura.

```
roservicioSpringBootCoreDeTiendaOnlineApplication.java   HolaMundoRestController.java
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HolaMundoRestController {

    public String saludo(){
        return "Hola Mundo Spring Boot";
    }

}
```

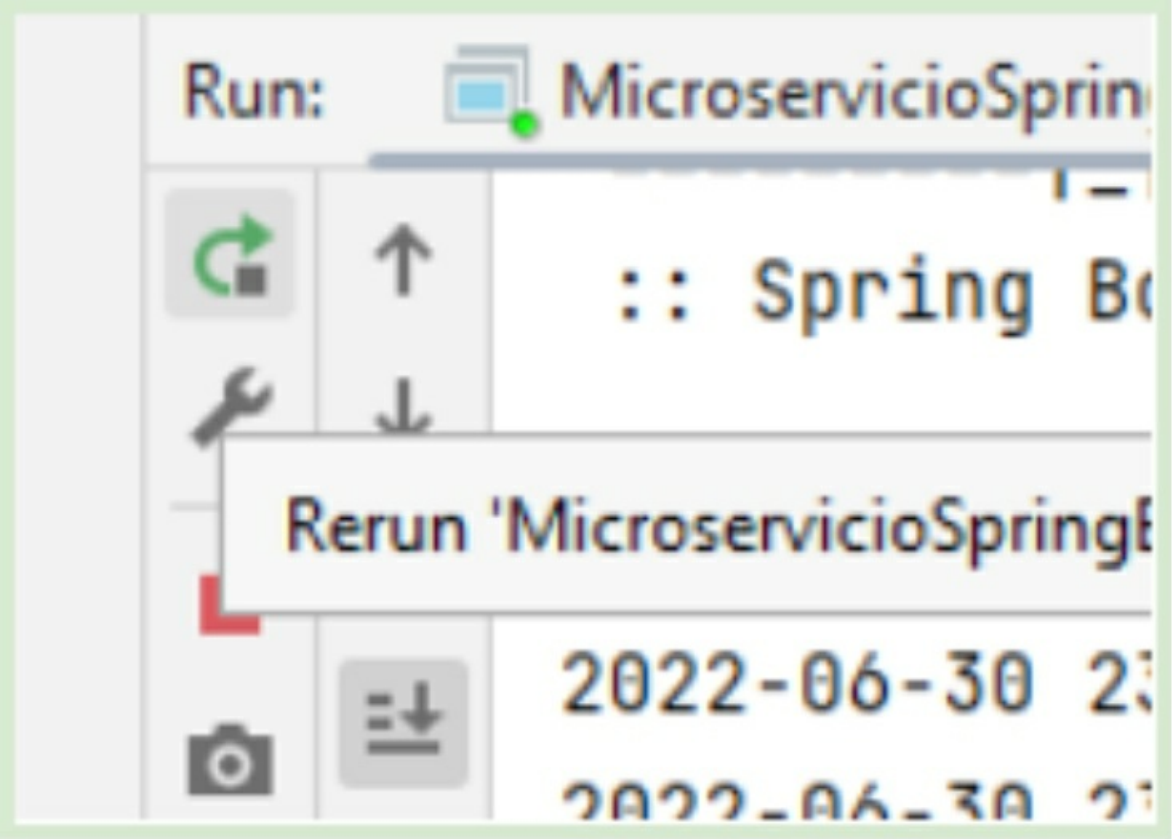
Esta anotación de Spring indica que esta clase Java oficiara de controlador y expondrá varios endpoints de tipo REST, que serán los métodos de esta clase. Es así que el método "saludo" será un endpoint web REST que al consumirlo simplemente responderá con un típico saludo de "Hola Mundo".

Para lograr que el método *"saludo"* sea un servicio REST, tendremos que decorarlo con la anotación @GetMapping("/saludo")

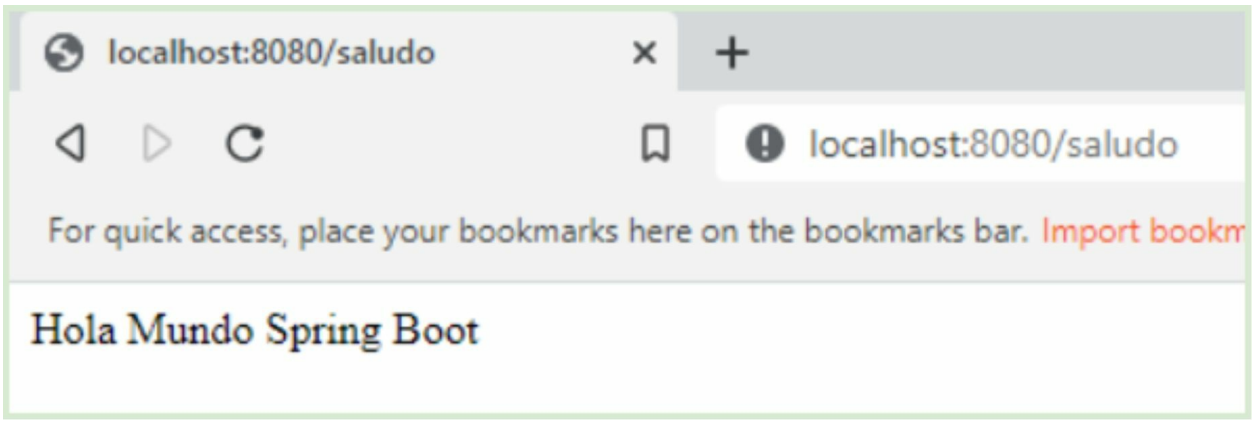
```
roservicioSpringBootCoreDeTiendaOnlineApplication.java   HolaMundoR
import org.springframework.web.bind.annotation.Rest

@RestController
public class HolaMundoRestController {

    @GetMapping("/saludo")
    public String saludo(){
        return "Hola Mundo Spring Boot";
    }

}
```

Con esta anotación indicamos que este método es un servicio REST que se expondrá mediante el método HTTP GET, que es el método HTTP más frecuente. Y finalmente le indicamos que este servicio podrá ser consumido bajo el nombre *"saludo"*. En este caso el nombre puede ser cualquier otro, incluso distinto al nombre del método en sí.

¡Ahora sí! tenemos todo listo para ejecutar nuestra primera clase de tipo controlador en Spring. Debemos ejecutar nuevamente el proyecto. En el caso de que lo hayan dejado ejecutando, lo tendrán que reiniciar haciendo click en el icono "Rerun" de la solapa "Run" en la parte inferior del IntelliJ.



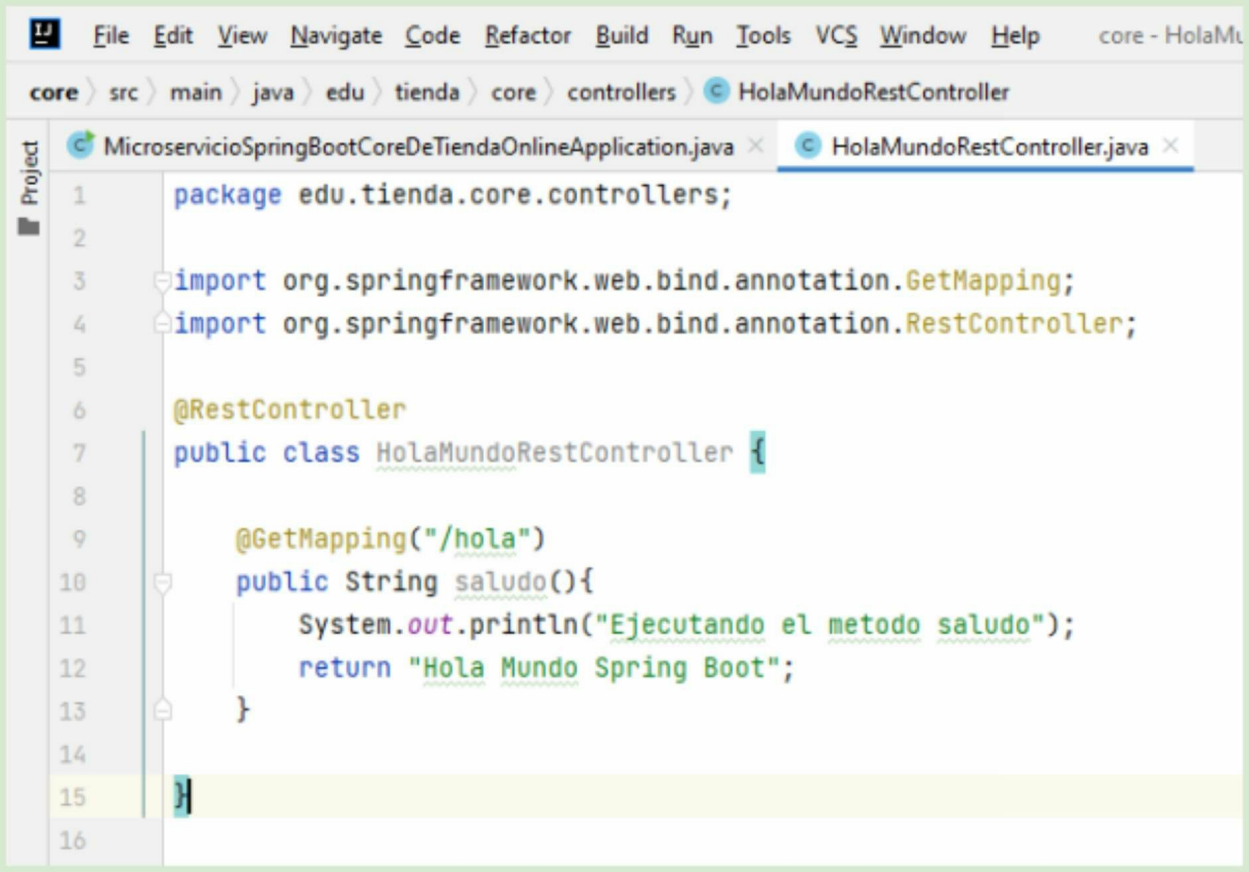
Para probar nuestra clase y saludo, simplemente abriremos un navegador web cualquiera y escribiremos la dirección http://localhost:8080/saludo



¡Felicitaciones!, Hemos escrito nuestro primer controlador y servicio REST en Spring Boot. Observemos que al atacar http://localhost:8080, estamos apuntando al servicio Tomcat embebido que levanta Spring Boot al iniciar el proyecto.

En nuestra URL hemos indicado con el sufijo "/saludo" que queremos consumir el servicio web de tipo REST que está expuesto mediante el método "*saludo*" creado en nuestra clase java. Este método retorna una simple cadena de caracteres que es la que presenta el explorador en la vista de página.

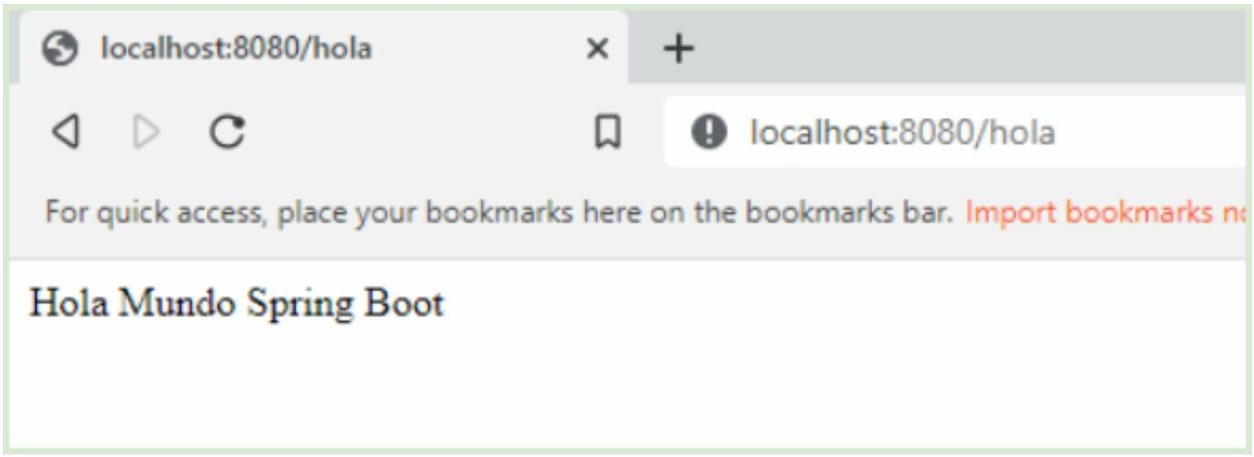
Ahora vamos a jugar un poco con el código y realizar algunos cambios como se propone a continuación:



Como verán, hemos realizado dos pequeñas modificaciones. La primera es que hemos cambiado el nombre del endpoint REST por la cadena *"hola" (línea número 9)* y el segundo es que agregamos un *"println"* a forma de trazabilidad que mostrará un mensaje en consola cada vez que se ejecute el endpoint en cuestión.

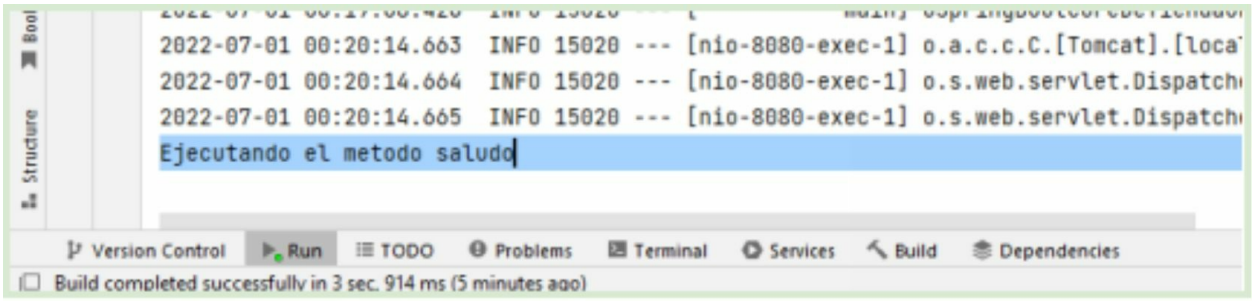
Es importante mencionar que cada cambio en el código fuente del proyecto requiere un reinicio de Spring para que esos cambios se hagan efectivos. Existe un plugin de Spring Boot que permitirá realizar modificaciones "en caliente" evitando este reinicio. Se llama Spring Boot Devtools y pueden agregarlo como dependencia en maven para evitar los reinicios de Spring por cada cambio efectuado en el código fuente.

Una vez reiniciado, podremos consumir el mismo endpoint pero ahora apuntando a la siguiente dirección web http://localhost:8080/hola



Este pequeño cambio en el nombre del endpoint es para demostrar que el nombre del servicio REST expuesto y consumible desde el mundo exterior (en este caso desde un navegador) es proveído por el decorador @GetMapping("/hola") y no por el nombre de firma del método que es *"saludo"*.

El segundo cambio nos permitirá evidenciar que cada vez que el servicio REST es consumido, se ejecuta el método *"saludo"* que emite una traza en la consola del IDE.



Capítulo 3

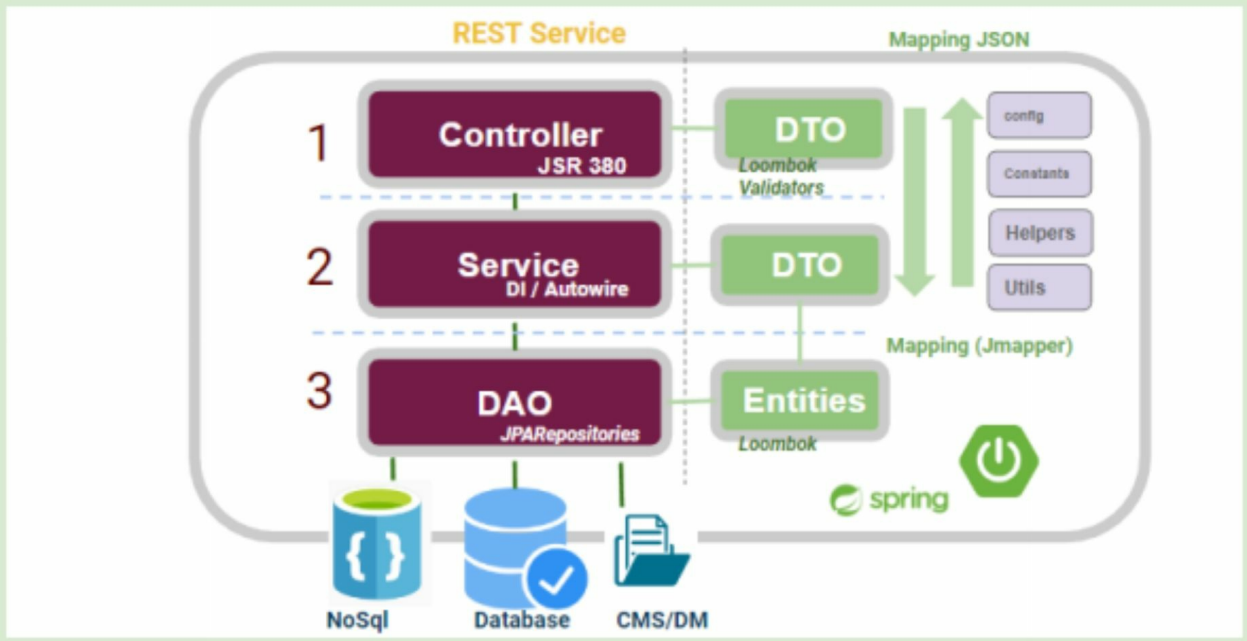
3.1 Capa controller

Durante este capítulo nos enfocaremos exclusivamente en la capa controller.

3.1.1 Arquitectura basada en capas

En nuestra arquitectura basada en tres capas, nos vamos a enfocar exclusivamente en la capa de “nivel superior”, que es la denominada capa controladora. Esta capa como otras está definida en el paquete controller y está compuesta de un conjunto de clases javas que la constituyen.

En un sistema de tipo “Back End” como el que estamos construyendo, la capa controller es la que tiene la responsabilidad de “dialogar” con el “mundo exterior”. Es decir, con los demás sistemas externos que van a usufructuar este artefacto de software. Cuando nos referimos a sistemas externos estamos hablando de uno o varios frontales web que pueden formar o no parte del stack de nuestra solución, o también, a diferentes clientes como un explorador web o un sistema REST especializado como POSTMAN que veremos más adelante en este libro.



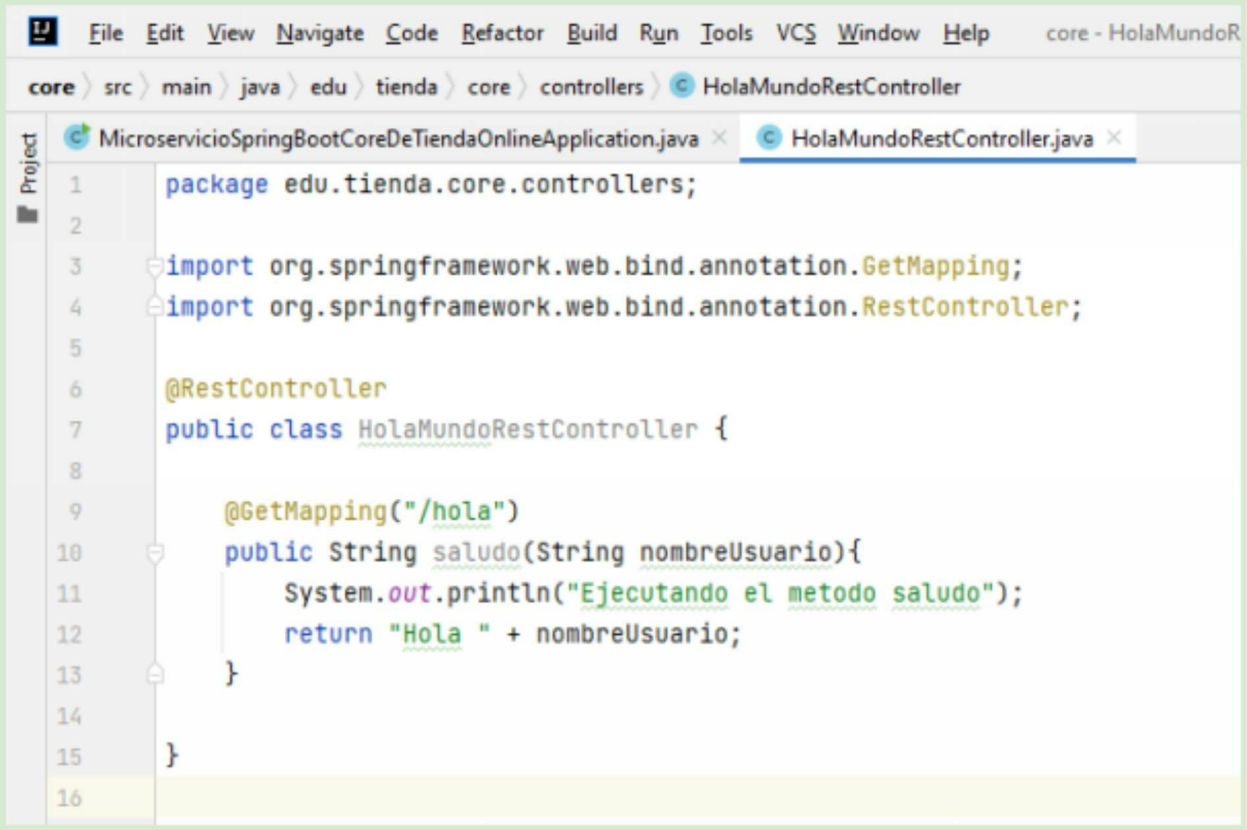
En la infografía se pueden apreciar en violeta las tres capas que constituyen la arquitectura de nuestro microservicio back-end. Como se puede observar, la capa controller se dispone en la parte superior del artefacto.

La capa de controlador tiene el propósito de exponer los servicios de tipo REST, aunque vale aclarar que podrían presentarse como servicios WEB SOAP u otro tipo de estándar también. Este conjunto de servicios REST conforman lo que se denomina como la API de nuestro software, esto significa, será la interfaz de uso y comunicación que propone este artefacto de Back End.

3.1.2 Servicio Rest con parámetros

Durante esta sección, iremos descubriendo las diferentes anotaciones que Spring nos ofrece para potenciar esta capa y aprenderemos cómo implementar una API REST bien formada. Ahora sí retomaremos el método “saludo” que hemos implementado para incorporar la posibilidad de que este saludo sea personalizado.

Para lograr que ese saludo sea personalizado, expondremos el mismo endpoint REST pero ahora con la posibilidad de que el consumidor pueda proporcionar un nombre de usuario. Para ello vamos a modificar el código fuente como se aprecia en la siguiente imagen.



Estos primeros cambios fueron exclusivamente en “java plano”. Simplemente hemos agregado un parámetro al método saludo de tipo *“String” en la línea número 10* que contendrá el nombre del usuario. Como pueden observar, hemos modificado también el saludo para que pueda ser personalizado en la línea número 12.

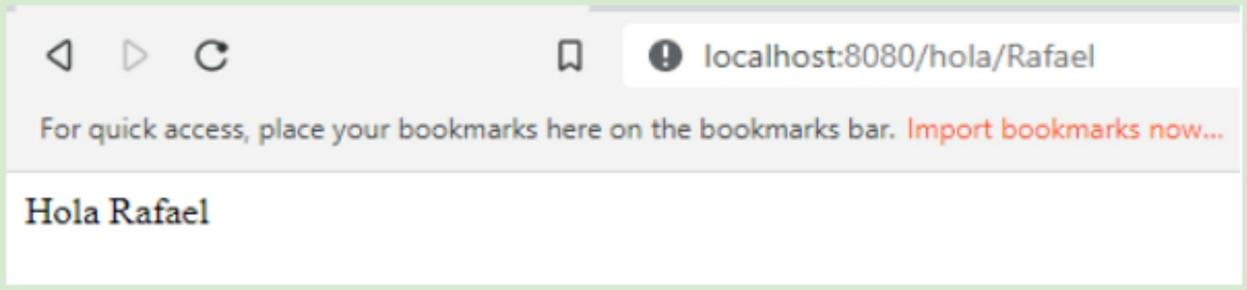
Con estos cambios aún no basta para que la api *“/hola”* pueda recibir un parámetro, sino que para ello, deberemos realizar unos simples agregados utilizando el potencial de Spring.

1. Decoramos el parámetro “String *nombreUsuario”* con la anotación @PathVariable
2. En segundo lugar, modificaremos el nombre del endpoint para indicarle que este puede recibir un parámetro *“{nombreUsuario}”* tal como se muestra a continuación.

```java
@RestController
public class HolaMundoRestController {

    @GetMapping("/hola/{nombreUsuario}")
    public String saludo(@PathVariable  String nombreUsuario){
        System.out.println("Ejecutando el metodo saludo");
        return "Hola " + nombreUsuario;
    }

}
```

Nuestro código fuente ahora luce como se puede observar en la imagen superior. En principio hemos "decorado" el parámetro de tipo "String" con una anotación Spring llamada @PathVariable que indica que este parámetro java normal se transformará en un parámetro web de tipo Path Variable, es decir, un parámetro de tipo web que estará contenido en la URL del endpoint.

Reiniciamos el proyecto Spring para probar nuestra nueva funcionalidad y podremos consumir el endpoint REST, en esta oportunidad, atacando la siguiente URL: http://localhost:8080/hola/**Rafael**



Como se puede observar, nuestra URL contiene un sufijo que es el valor del parámetro que suministramos. En otras palabras, estamos ejecutando el mismo servicio REST pero ahora le proporcionamos un parámetro de tipo *"String" que es "Rafael"*. El método java *"saludo"* finalmente utiliza este parámetro para devolver un saludo más personalizado.

3.2 Capa controller - CRUD Cliente

En esta sección implementaremos un api REST de alta, baja y modificación de cliente

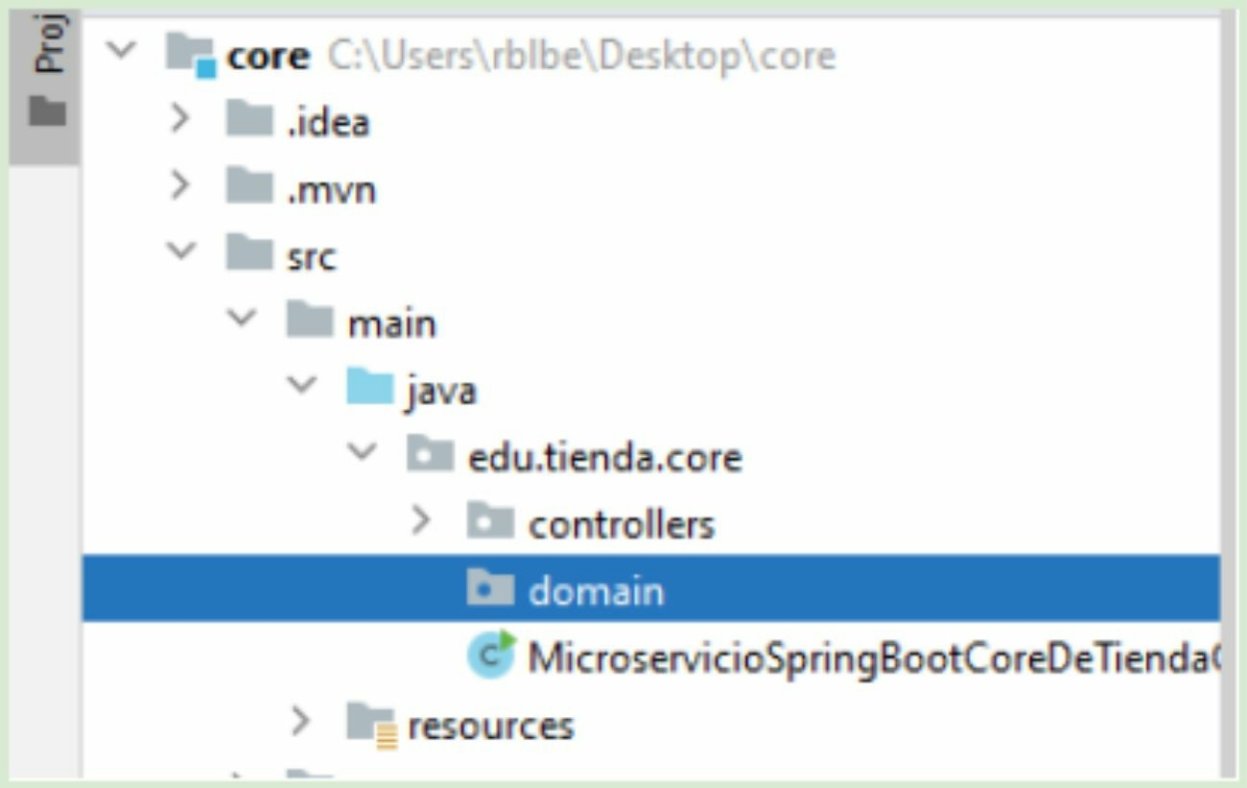
3.2.1 Implementando un API Rest

Ahora que hicimos un acercamiento a cómo implementar un servicio REST básico, vamos a profundizar en Spring para implementar una API más parecida al mundo real. En este caso implementaremos un CRUD de clientes. Es decir, una API REST que permite la alta, baja, modificación y eliminación de clientes de nuestro sistema.

1. ***Generar la clase de dominio***

Antes de comenzar a implementar el nuevo controlador, crearemos una clase java "POJO". Se trata de una clase básica de dominio o modelo que simplemente contendrá atributos básicos y los respectivos métodos "getter" y "setter" para acceder a estos.

1. Creamos el paquete *"domain"* dentro del paquete *"edu.tienda.core"*

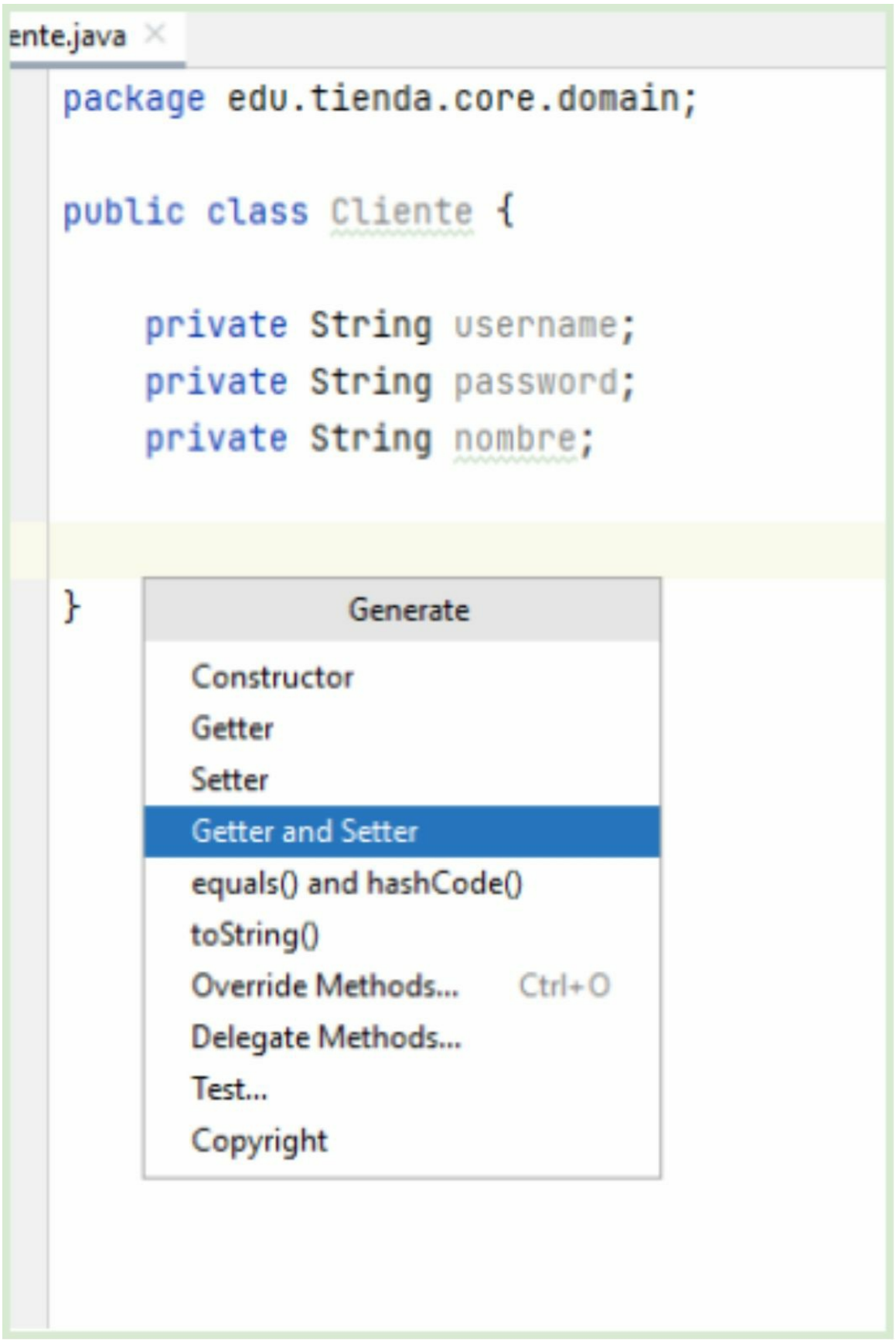


2. Dentro del paquete *"domain"* creamos nuestra nueva clase *"Cliente"*

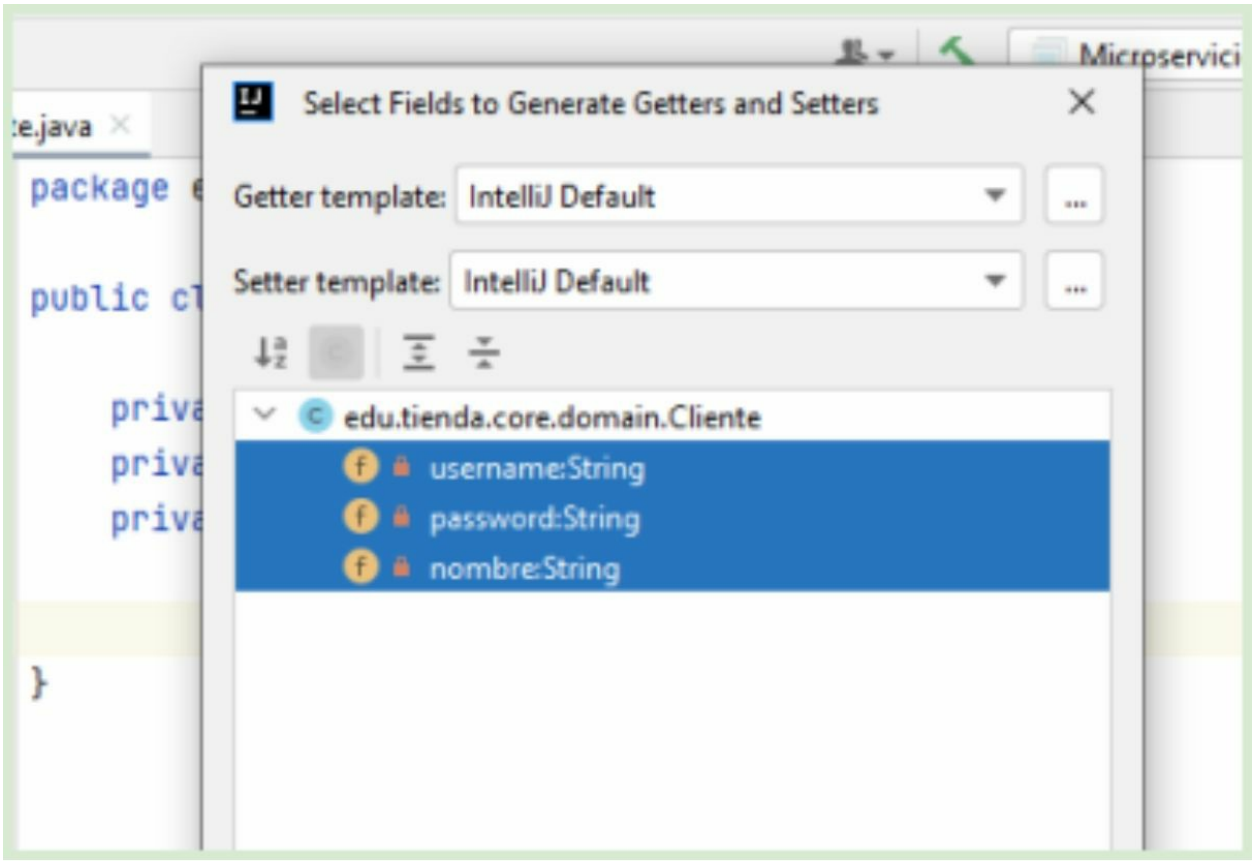
```java
package edu.tienda.core.domain;

public class Cliente {

    private String username;
    private String password;
    private String nombre;

}
```

3. Hemos definido nuestra clase *"Cliente"* con tres atributos de tipo *"String"*. Ahora es necesario generar los respectivos métodos de acceso para cada uno de ellos. Para esto nos vamos a ayudar con IntelliJ.

1. Sobre el cuerpo de la clase y debajo del atributo nombre presionamos "ALT+INS" para que el IDE nos presente el siguiente menú emergente.


ente.java
package edu.tienda.core.domain;
public class Cliente {
private String username;
private String password;
private String nombre;
}
Generate
Constructor
Getter
Setter
Getter and Setter
equals() and hashCode()
toString()
Override Methods... Ctrl+O
Delegate Methods...
Test...
Copyright

2. Hacemos click en “Getter and Setter” y se nos presenta el siguiente asistente.



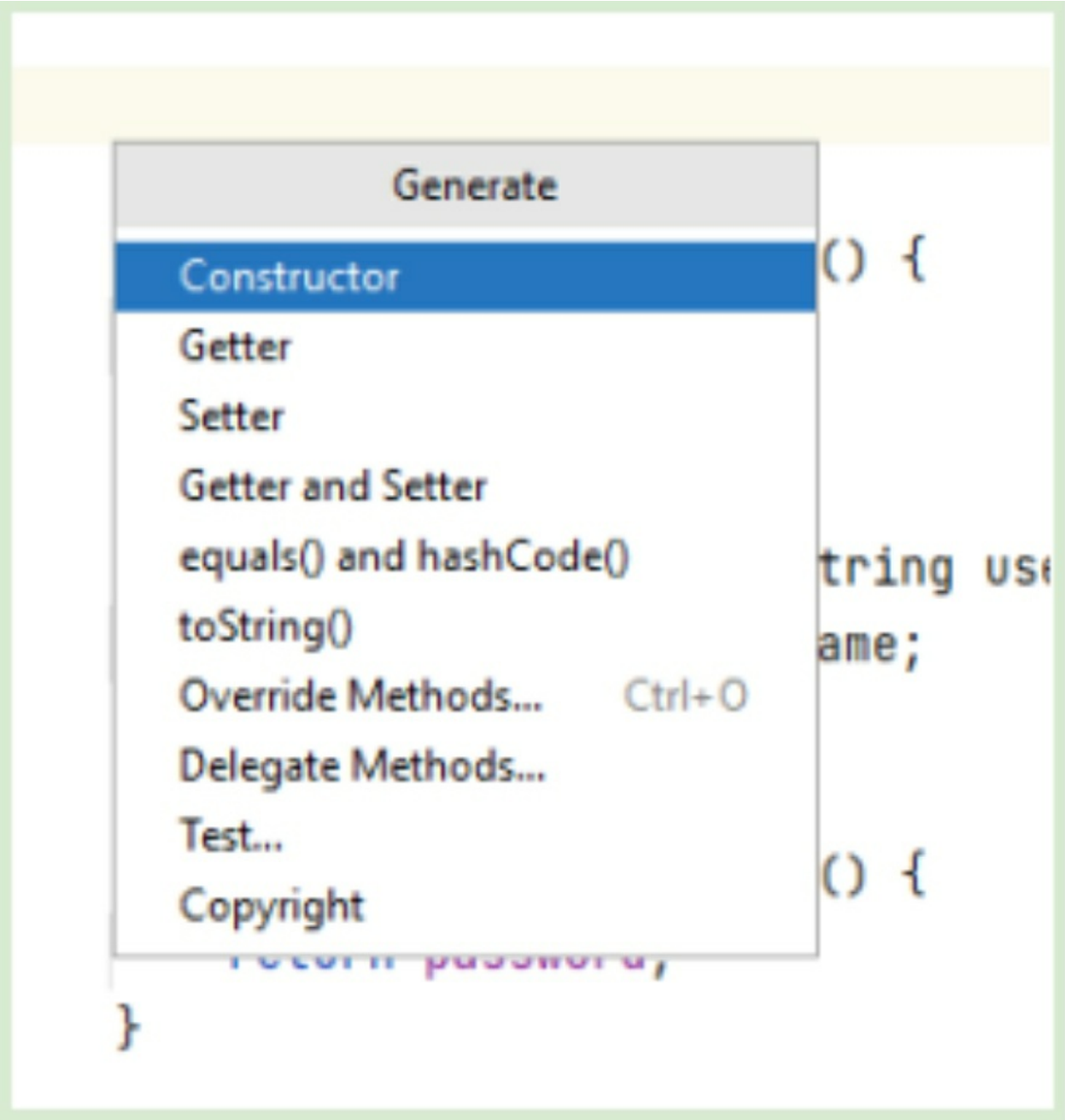
3. Seleccionamos los tres atributos y presionamos el botón “ok”.

IntelliJ nos generará automáticamente los tres métodos “getter” y los tres métodos “setter” para manipular desde el exterior los atributos de la clase.

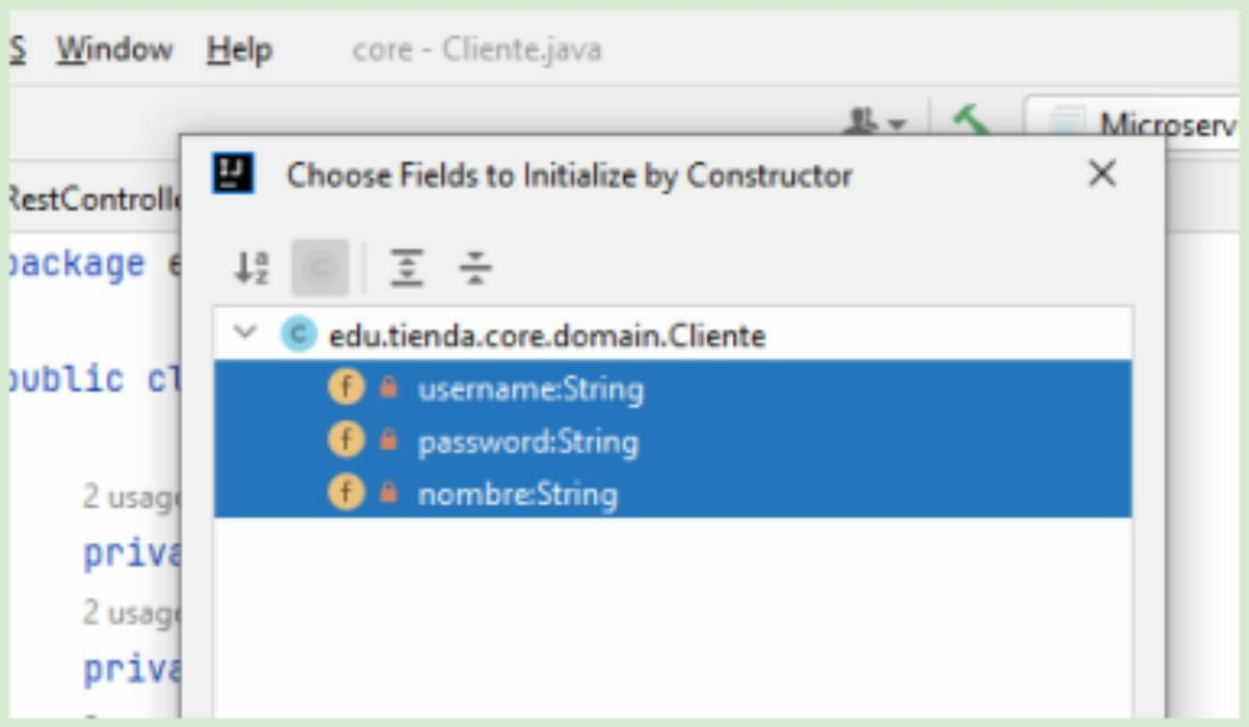
```
    private String nombre;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
```

4. También crearemos un constructor de clase apoyándonos con nuestro IDE. Presionamos nuevamente “ALT+INS” y seleccionamos la opción de menú

“Constructor”.



A continuación seleccionaremos los tres atributos para que el constructor pueda fabricar el objeto a partir de toda esta información.



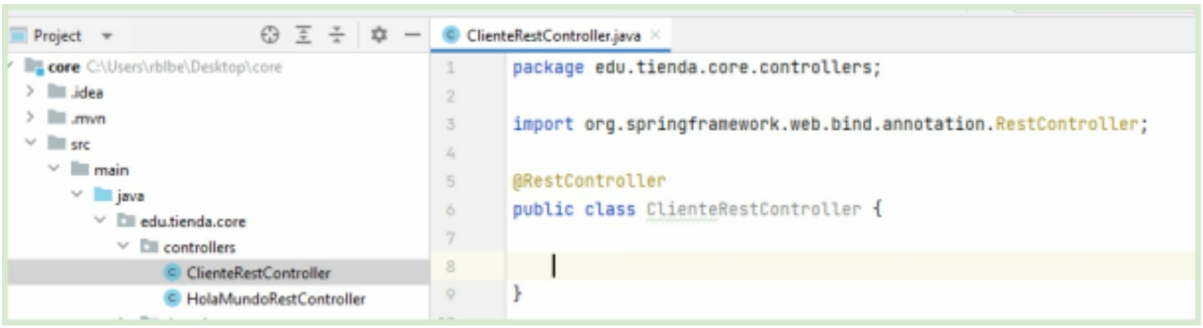
Finalmente nuestra clase tomará este aspecto.

```java
public class Cliente {

    3 usages
    private String username;
    3 usages
    private String password;
    3 usages
    private String nombre;

    public Cliente(String username, String password, String nombre) {
        this.username = username;
        this.password = password;
        this.nombre = nombre;
    }
```

Procederemos a crear nuestro nuevo controlador REST para el manejo de clientes donde esta clase contendrá la información de clientes que iremos transportando en cada petición de los servicios expuestos.

2. Generación del controlador de cliente

1. Nos posicionamos nuevamente en el paquete *"edu.tienda.core.controller"* y creamos una clase java con el nombre *"ClienteRestController"*.
2. Anotamos la clase con el decorador @RestController como ya lo hemos hecho con la clase *"HolaMundoRestController"*.



3. Nuestra clase Java ya se ha transformado en un controlador de tipo dREST y tenemos todo listo para comenzar a incorporar cada uno de los métodos para el alta, baja, modificación y eliminación.

3. Simulación de clientes en memoria

1. Para que nuestra API sea un poco más realista, crearemos un almacén de clientes en memoria. Esto lo implementaremos con una simple lista que simulara ser una especie de base de datos de clientes.

```
@RestController
public class ClienteRestController {

    3 usages
    private List<Cliente> clientes = new ArrayList<>(Arrays.asList(
            new Cliente( username: "arm", password: "1234", nombre: "Armstrong"),
            new Cliente( username: "ald", password: "1234", nombre: "Aldrin"),
            new Cliente( username: "col", password: "1234", nombre: "Collins")
        )
    );
```

En el código anterior se puede observar que hemos creado una lista de clientes en un solo "Shoot". Esta lista de prueba emplea el rol de persistencia o almacén de clientes en nuestra API. A continuación, Implementaremos el primer método del CRUD de clientes.

2. Obtención de todos los clientes del sistema.

Implementaremos un endpoint que nos devolverá todos los clientes de nuestro sistema. Este será un simple servicio REST de tipo GET. Cabe resaltar que el mecanismo HTTP GET suele y debe ser utilizado para servicios que solo recuperan información y no se debe emplear en aquellos servicios de creación o modificación del modelo de datos. Durante la creación de esta API ahondaremos más en estos métodos HTTP y su adecuado uso para cada servicio.

i. Generamos un método java con el nombre *"getClientes"* que retorne una Lista de clientes.

```
    public List<Cliente> getClientes(){

        return clientes;
    }
```

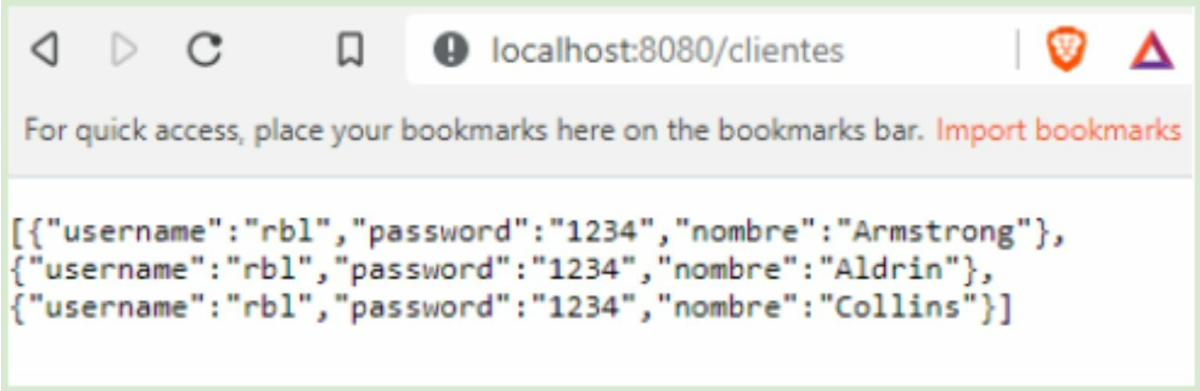
Como hemos comentado anteriormente, y hasta el momento, se trata de un simple método Java que retorna la lista de clientes que se genera automáticamente en el inicio de esta clase.

Es un simple método Java hasta que lo decoramos con la anotación @GetMapping que la convierte, ahora sí, en un endpoint REST que puede ser consumido desde el exterior.

```
@GetMapping("/clientes")
public List<Cliente> getClientes(){

    return clientes;
}
```

Nuestro método luce su flamante decorador que le permite exponerse al mundo en forma de servicio REST. Este endpoint puede ser consumido utilizando la dirección http://localhost:8080/clientes

Al reiniciar Spring Boot podremos probar el servicio desde un explorador y obtendremos el siguiente resultado.



En este punto nos detendremos para observar e interpretar la respuesta que hemos conseguido. Aunque nos resulte la respuesta un tanto lógica e intuitiva, deberemos apreciar que Spring ha realizado una serie de procedimientos de forma automática a los cuales nos abstraemos totalmente.

En principio y antes de retornar el cuerpo de respuesta, Spring, ha convertido la lista de clientes (*List<Clientes>*) a un array de tipo JSON. Es decir, ha realizado por debajo un minucioso proceso de serialización desde JAVA a JSON. En otros frameworks más antiguos esta era una ardua tarea que le correspondía al desarrollador y era necesario lidiar con sofisticados mappers de JSON o XML.

3. Obtención de un cliente por *"userName"*

Ahora implementaremos un método para recuperar un cliente por su nombre de usuario. Para

esto deberemos generar un método con el siguiente formato.

```
    @GetMapping("/clientes/{userName}")
    public Cliente getCliente(@PathVariable String userName){
        for (Cliente cli : clientes){
            if (cli.getUsername().equalsIgnoreCase(userName)){
                return cli;
            }
        }
        return null;
    }

}
```

Hemos implementado un método que simplemente itera la lista de clientes hasta que encuentra una coincidencia de manera flexible por el nombre de usuario. En el caso de que el nombre de usuario no se encuentre, usaremos lo que sería una pésima práctica que es devolver *"null"*. Pero lo haremos para no sobrecargar de contenidos esta unidad ya que más adelante veremos cómo tratar este tipo de excepciones.

Es muy importante observar con lupa que el nombre del endpoint es exactamente igual al anterior servicio que retornaba a todos los clientes. No obstante hay un simple detalle que hace que estos dos servicios REST no tengan exactamente la misma denominación. Ese detalle que los hace diferentes, es que el último servicio que implementamos recibe un parámetro que está definido como "path variable" en el sufijo de la URL.

/clientes
/clientes/{userName}

Si bien el prefijo de la URL es igual (*"/clientes"*), el segundo endpoint se compone de un "sub-path" extra que será el valor del "userName" que se desea buscar. Estas pequeñas diferencias hacen que los servicios de una API-REST tengan un comportamiento parecido a lo que llamamos los métodos Java sobrecargados. Para entender este concepto, imaginemos que tenemos una clase Java con dos métodos:

public Cliente getCliente()
public Cliente getCliente(String name)

En este caso, como bien sabemos, se trata de dos métodos que comparten el mismo nombre de firma pero que son distintos. Y esa distinción se logra gracias a que el segundo método recibe un parámetro y el primero no. Este tipo de definiciones las conocemos como

“sobrecarga” de métodos Java (“overloading”) y es exactamente el mismo que se puede aplicar en una API REST.

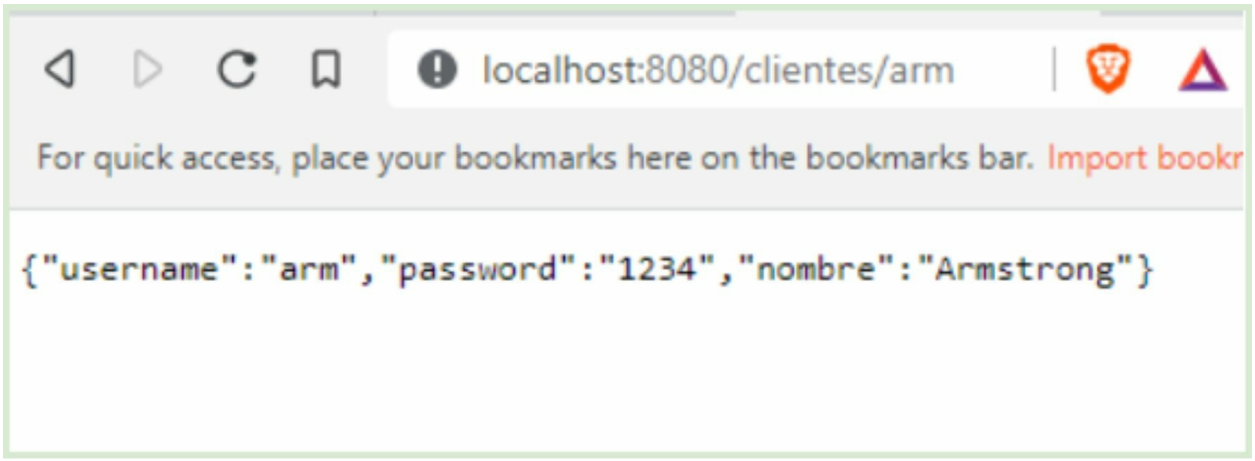
¡Importante!, también observamos que la firma de los endpoints definidos llevan el nombre “clientes” que es un sustantivo y plural. Se recomienda seguir estas reglas gramaticales para la definición de API’s REST, ya que este subprotocolo está orientado a recursos y no a acciones. Es por eso que debemos evitar definir nombres como “getCliente” o “recuperarClientes” para la definición de endpoints.

Volviendo a nuestro método de recupero de cliente por “userName”, ahora presentaremos una manera alternativa de implementar el mismo algoritmo de búsqueda, pero esta vez orientado al uso de streams combinado con funciones lambda, lo que transforma nuestro código en mas elegante y sintáctico.

```
@GetMapping("/clientes/{userName}")
public Cliente getCliente(@PathVariable String userName){
    return clientes.stream().
            filter(cliente -> cliente.getUsername().equalsIgnoreCase(userName)).
            findFirst().orElseThrow();
}
```

Observamos que con esta nueva codificación tuvimos que emitir una excepcion general en el caso que no se encuentre el cliente solicitado en la lista. Este código convierte a la lista en un “Stream”, que nos permitirá manipularla de manera más ligera, utilizando en este caso el método lambda *“filter”* donde ubicamos la lógica de búsqueda.

Reiniciamos Spring y atacamos el nuevo servicio con el explorador apuntando a la dirección http://localhost:8080/clientes/arm



Como observamos, el servicio ha retornado un objeto JSON que representa al cliente cuyo *“username”* es *“arm”*, que es el valor del parámetro que hemos suministrado en el sufijo de la URL.

3.3 Capa controller - CRUD Cliente - Modificación

En esta sección implementaremos los métodos que efectuaran una escritura o alteración en nuestro modelo de datos.

3.3.1 Implementando un alta de cliente

Una acción de alta o registro de nuevo cliente, a diferencia de los anteriores servicios de recupero, requiere una manipulación del modelo de datos. Es decir, que son acciones que modifican la lista o almacén de usuarios que hemos definido en nuestro controlador.

En el "idioma" REST, un endpoint de alta de recurso (en este caso un cliente) es un servicio que debe atacarse mediante el método HTTP POST. Esto no significa que no funcione si el servicio fuera definido como GET, sino que por convenciones y buenas prácticas se asocia con POST a todo aquel servicio que crea un nuevo recurso en nuestro sistema, y por su naturaleza, estos servicios requieren una entrada de información que debe ser proveída por el usuario que lo consume.

1. Implementaremos un simple método java con el nombre *"altaCliente"* que reciba un cliente y devuelva también una clase del mismo tipo.

```
public Cliente altaCliente(Cliente cliente){
    clientes.add(cliente);
    return cliente;
}
```

Simplemente este método recibe como parámetro un objeto de la clase *"Cliente"* y lo agrega a nuestra lista de almacén. Siendo un tanto repetitivos, esto se trata de un simplemente método java hasta que le realizamos las decoraciones apropiadas.

La primera anotación que incorporaremos es @PostMapping, que como pueden observar, es similar a @GetMapping ya que convierte a este método Java en un endpoint REST, pero en este caso mediante el mecanismo HTTP POST.

```java
@PostMapping("/clientes")
public Cliente altaCliente(Cliente cliente){
    clientes.add(cliente);
    return cliente;
}
```

Observar nuevamente que el nombre del endpoint *"/clientes"* es exactamente igual al resto de los endpoints definidos en esta clase, pero lo que los diferencia y sobrecarga de los demás, en este caso, es el método de acceso POST. Entonces a este punto tenemos tres endpoints con el mismo nombre pero diferenciados con las nomenclaturas resaltadas en naranja

GET /clientes
GET /clientes/username
POST /clientes

Ya hemos definido nuestro nuevo endpoint, sin embargo, todavía nos falta añadir una anotación para que el método permita la obtención de la clase *"Cliente"*. Esta anotación es @RequestBody.

```java
@PostMapping("/clientes")
public Cliente altaCliente(@RequestBody Cliente cliente){
    clientes.add(cliente);
    return cliente;
}
```

A diferencia de los anteriores servicios de recupero, este endpoint de alta lógicamente necesitará la información del usuario a crear. Esta información es proveída por el consumidor del servicio en formato JSON, y es Spring Boot, quien se encargará del trabajo de convertir ese JSON suministrado a un objeto de la clase *"cliente"*. Esta transformación es posible gracias a la anotación @RequestBody sobre el parámetro *"cliente"*, que lo que realmente está indicando es que este objeto se construirá a partir de la información enviada en el *"Body"* del *"Request"* para la petición de este servicio.

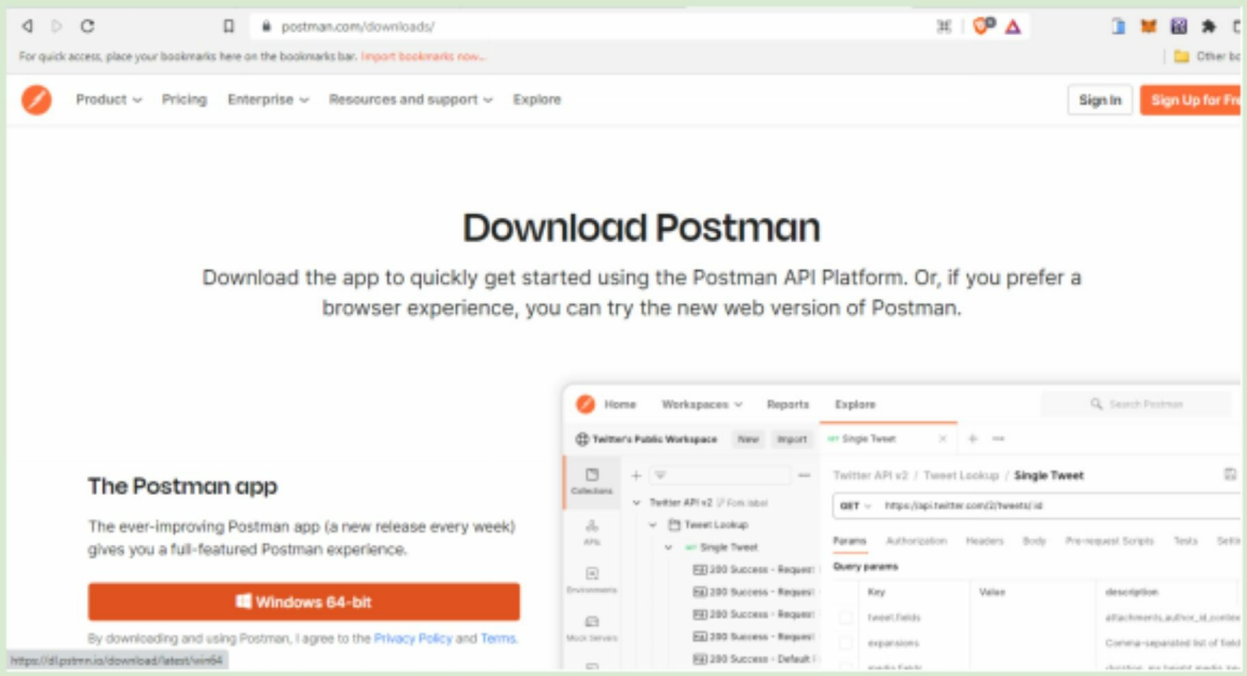
Ahora sí, todo listo para probar nuestro nuevo servicio REST. Pero en este caso nos vamos a topar con el siguiente inconveniente. Y es que los exploradores están preparados básicamente para consumir servicios o sitios webs mediante el mecanismo HTTP GET.

Cada vez que visitamos un sitio web como "google.com" o "amazon.com", por citar un

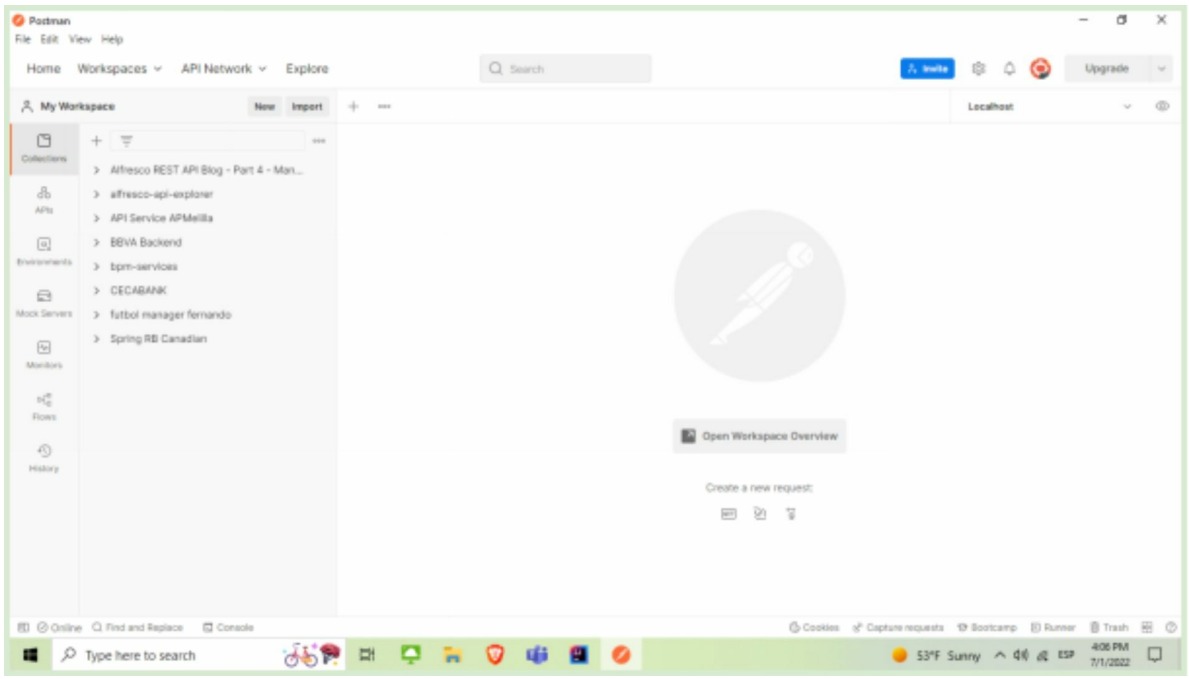
par de ejemplos, internamente nuestro explorador los resuelve utilizando la llamada GET. Algo así como GET http://google.com. Por esta razón es complicado poder consumir sitios o servicios que deben ser accedidos mediante POST utilizando un explorador WEB.

3.3.2 Instalando Postman para consumir API’s Rest

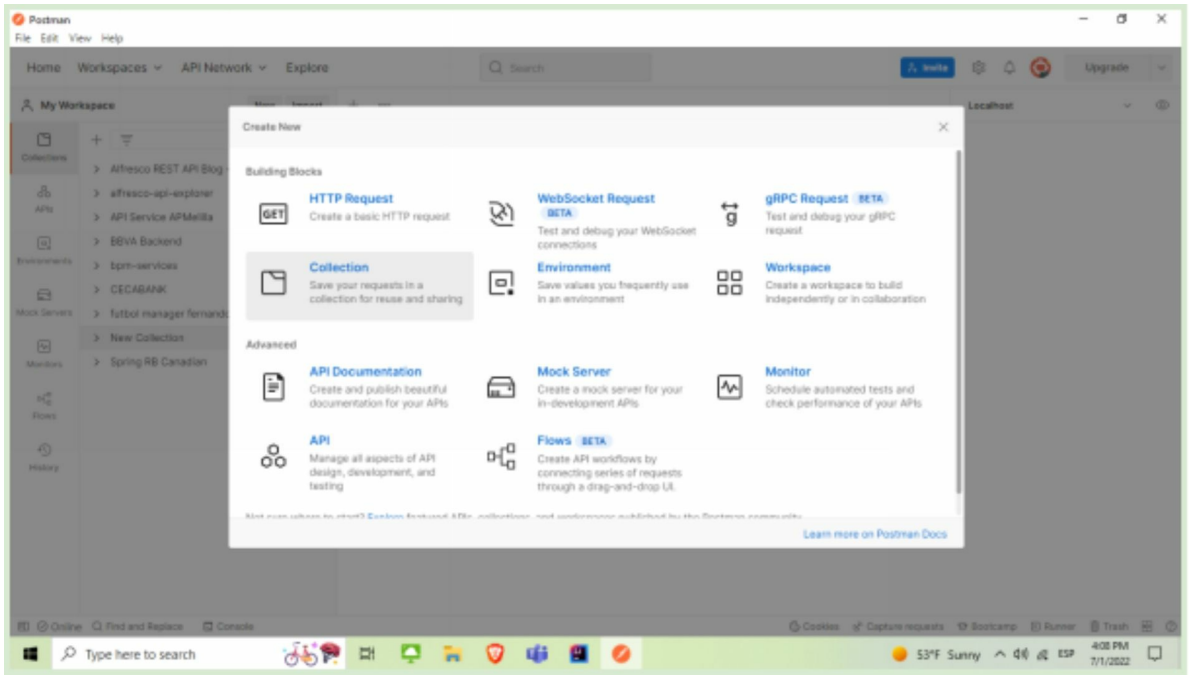
Para poder realizar este tipo de pruebas tendremos que ayudarnos con un software más especializado. Utilizaremos la famosa herramienta POSTMAN, que es una suite enfocada y dirigida para el consumo de servicios WEB o REST.Descargamos POSTMAN desde la direccion https://www.postman.com/downloads/



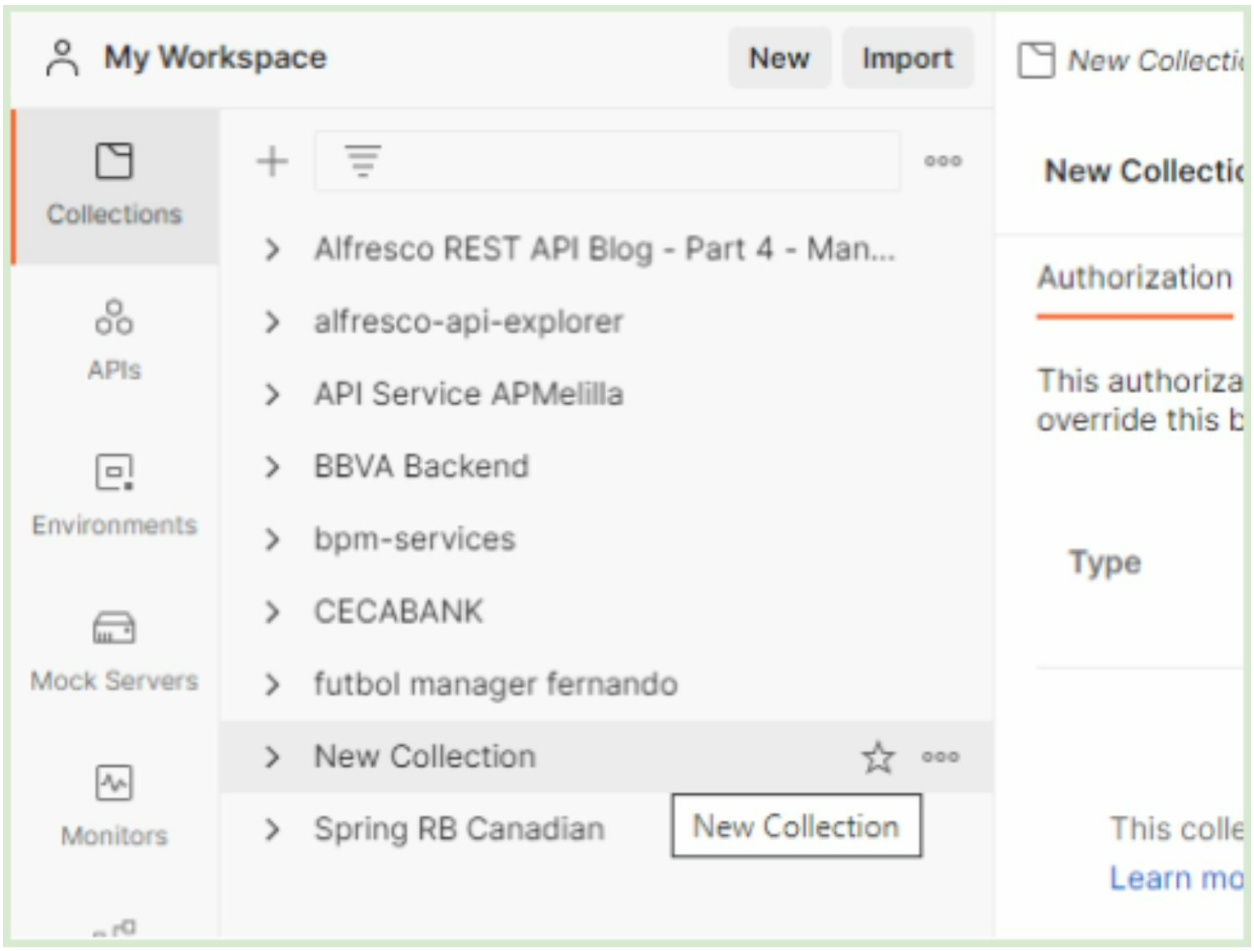
Postman se instala como cualquier programa básico y no requiere ningún tipo de configuración extra. Una vez instalado, podremos ejecutar POSTMAN en nuestra PC para realizar las pruebas pertinentes. Al ejecutar POSTMAN se nos presentará una pantalla similar a la siguiente



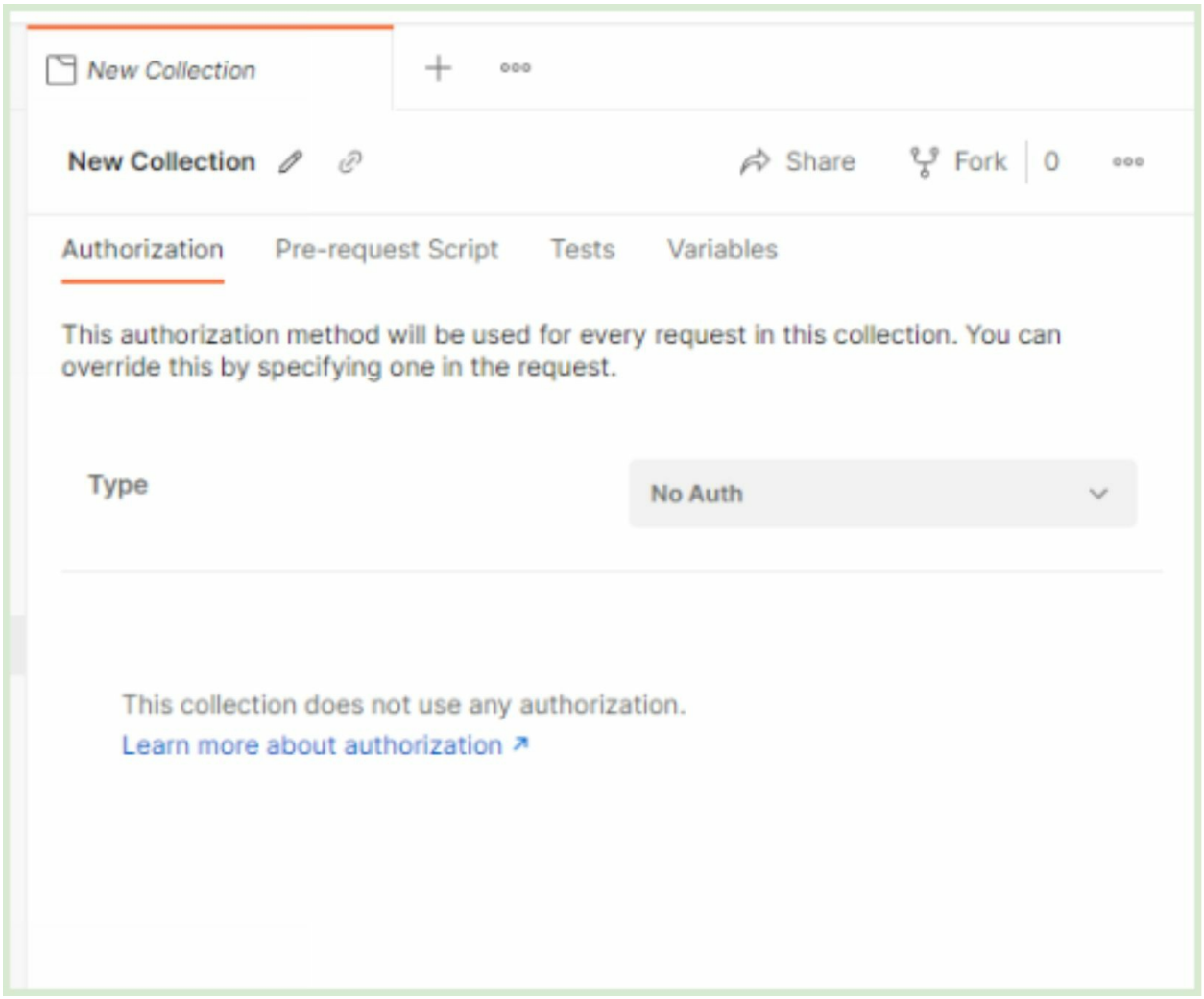
Para las pruebas de nuestras API Spring sugiero la creación de una colección “Postman” haciendo click en el botón “new” situado en la parte superior del frame izquierdo.



Al hacer click en “new” se nos presentará un menú donde seleccionaremos la opción “Collections” y se nos creará una nueva colección vacía como se muestra en la siguiente imagen.

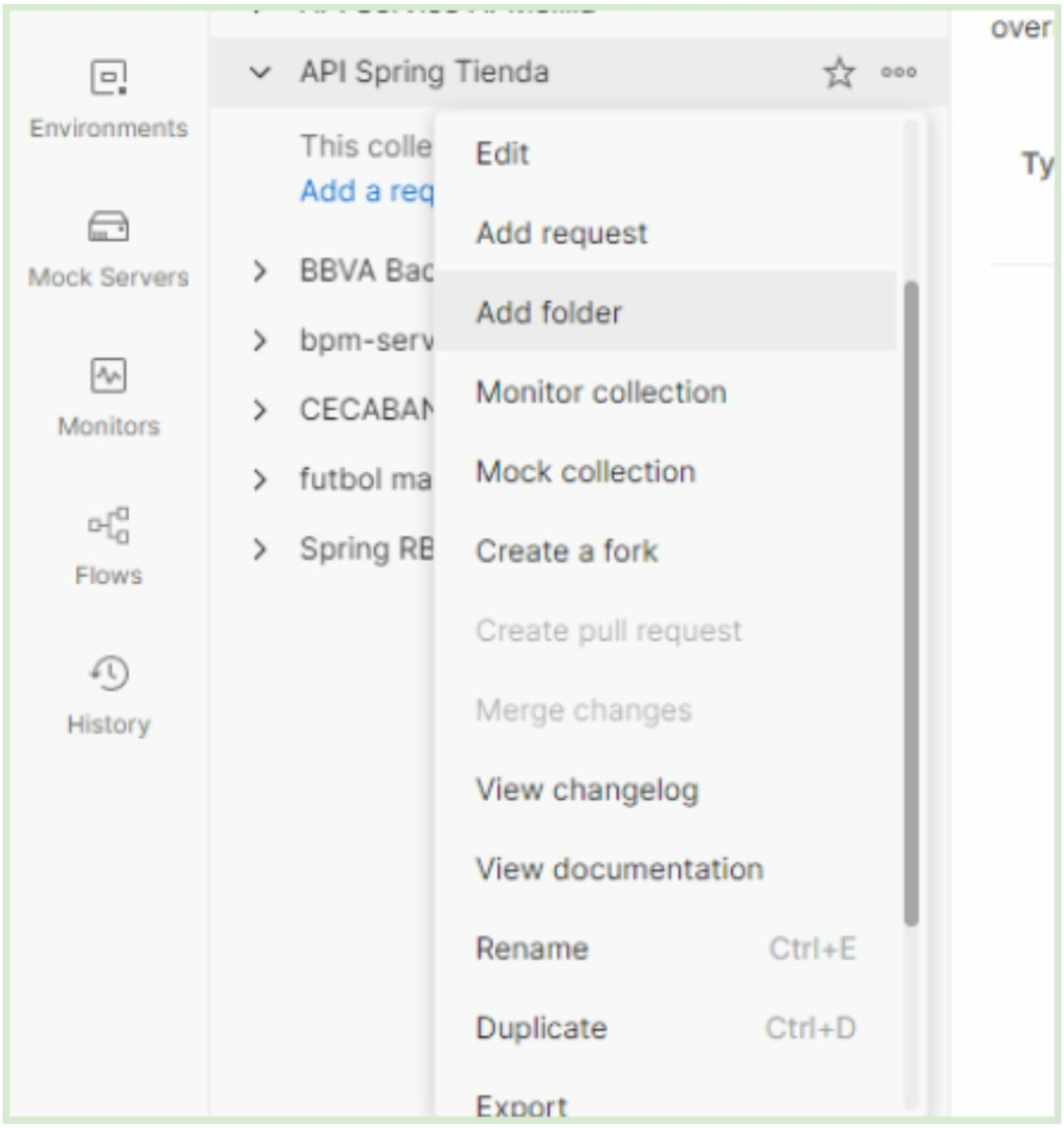


Esta colección, como se muestra en la imagen, se creará con el nombre por defecto “New Collection” en el frame izquierdo. Mientras que en el marco central se mostrará una carátula de la nueva colección donde podremos asignarle un nombre más atinado.

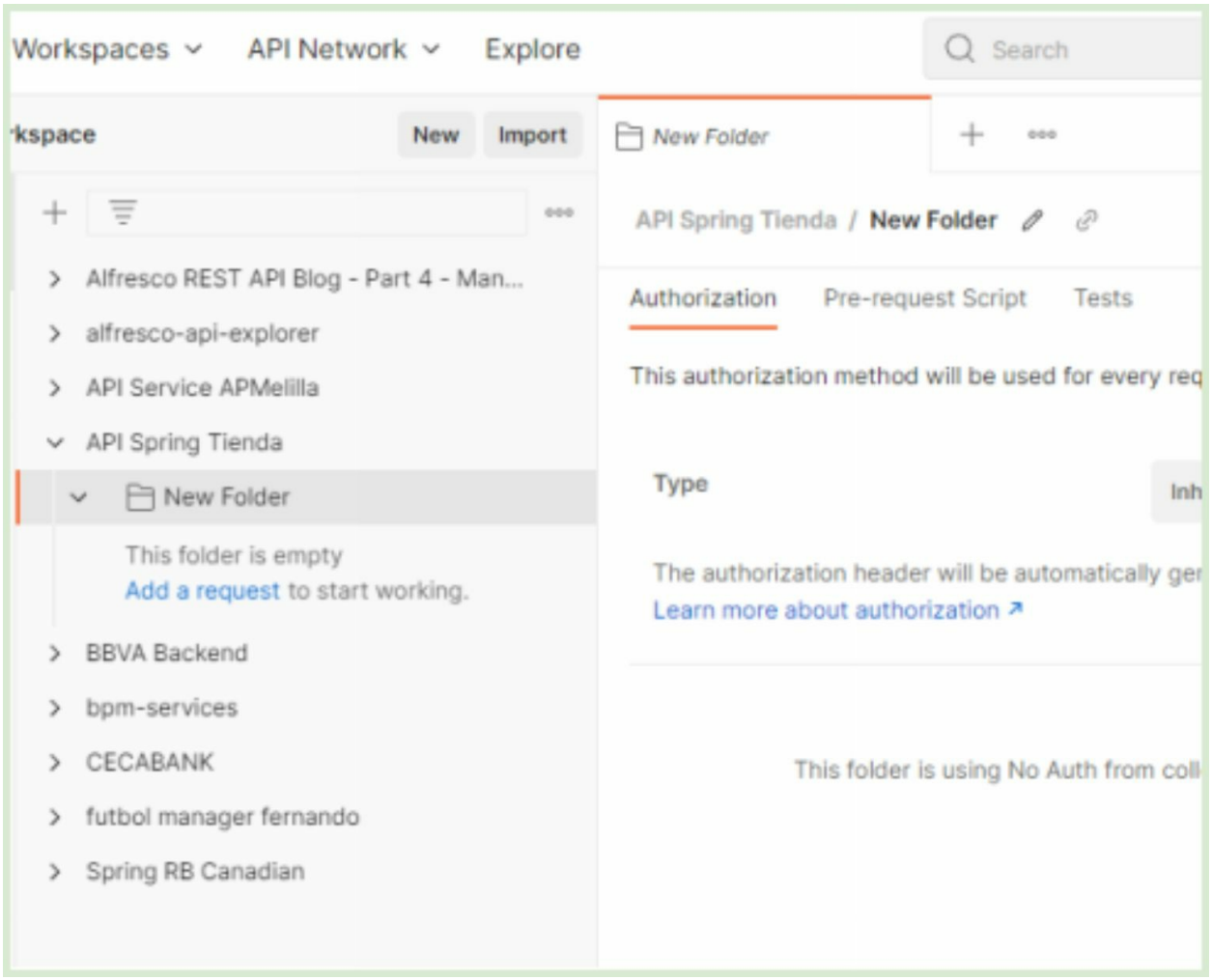


Haremos click sobre el icono del “lápiz” ubicado al lado del nombre de la colección y a continuación se abrirá un “input” que nos permitirá escribir el nuevo nombre. Le pondremos “API Spring Tienda”.

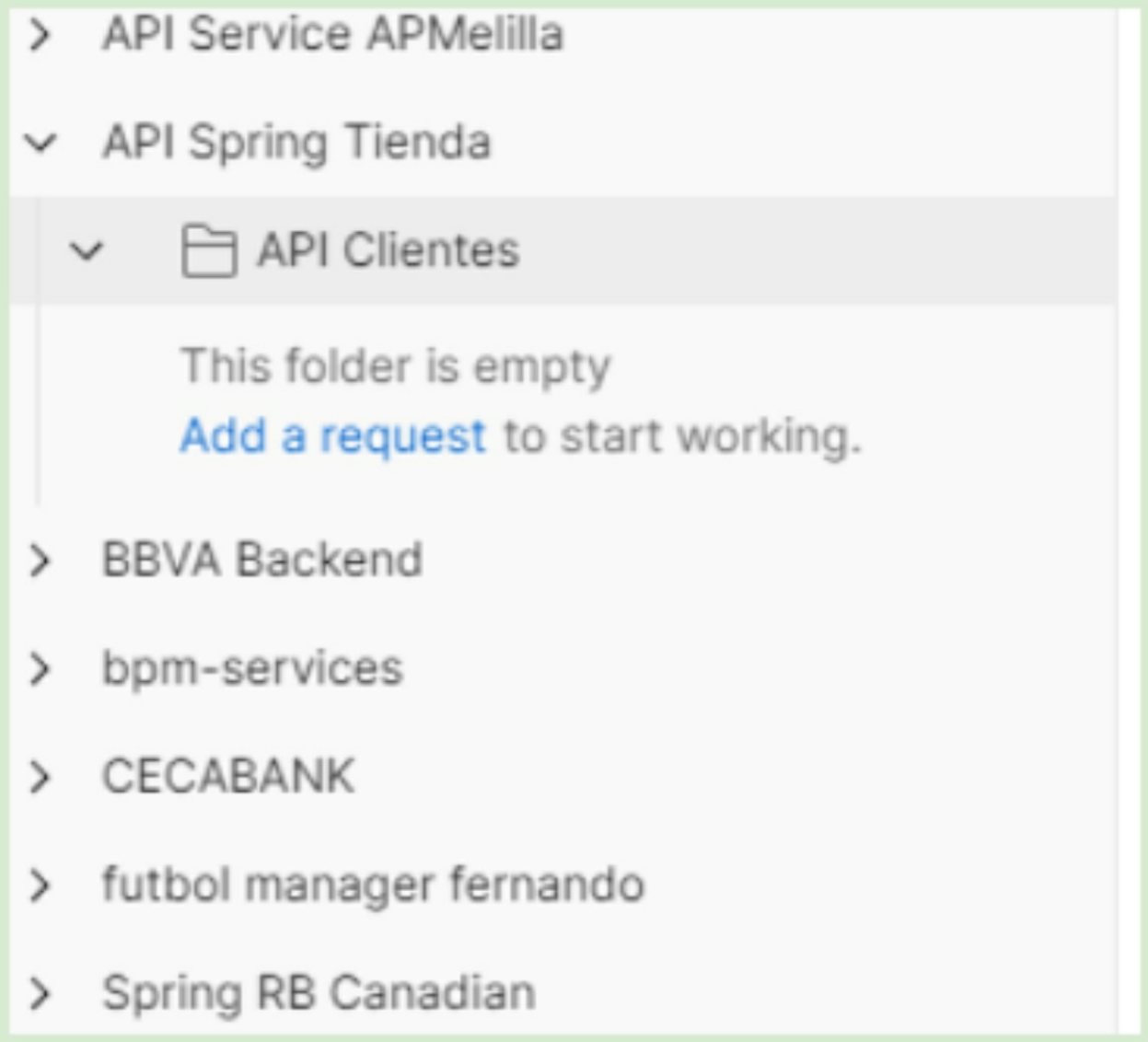
Luego de esta acción se renombrará la colección. Cabe mencionar que una colección de POSTMAN estará asociada con toda la API de nuestro proyecto y no sólo y exclusivamente con la “Sub-API” de gestión de clientes. Por lo tanto vamos a añadir una carpeta dentro de nuestra colección que tendrá el nombre “clientes”.



Para llevar a cabo esta acción hacer click con el botón derecho del mouse sobre el nombre de la colección exhibida en el marco izquierdo del programa. Se presenta un menú emergente y hacemos click en la opción “Add Folder”. Esta acción creará una nueva carpeta dentro de la colección con un nombre por defecto que también renombramos.



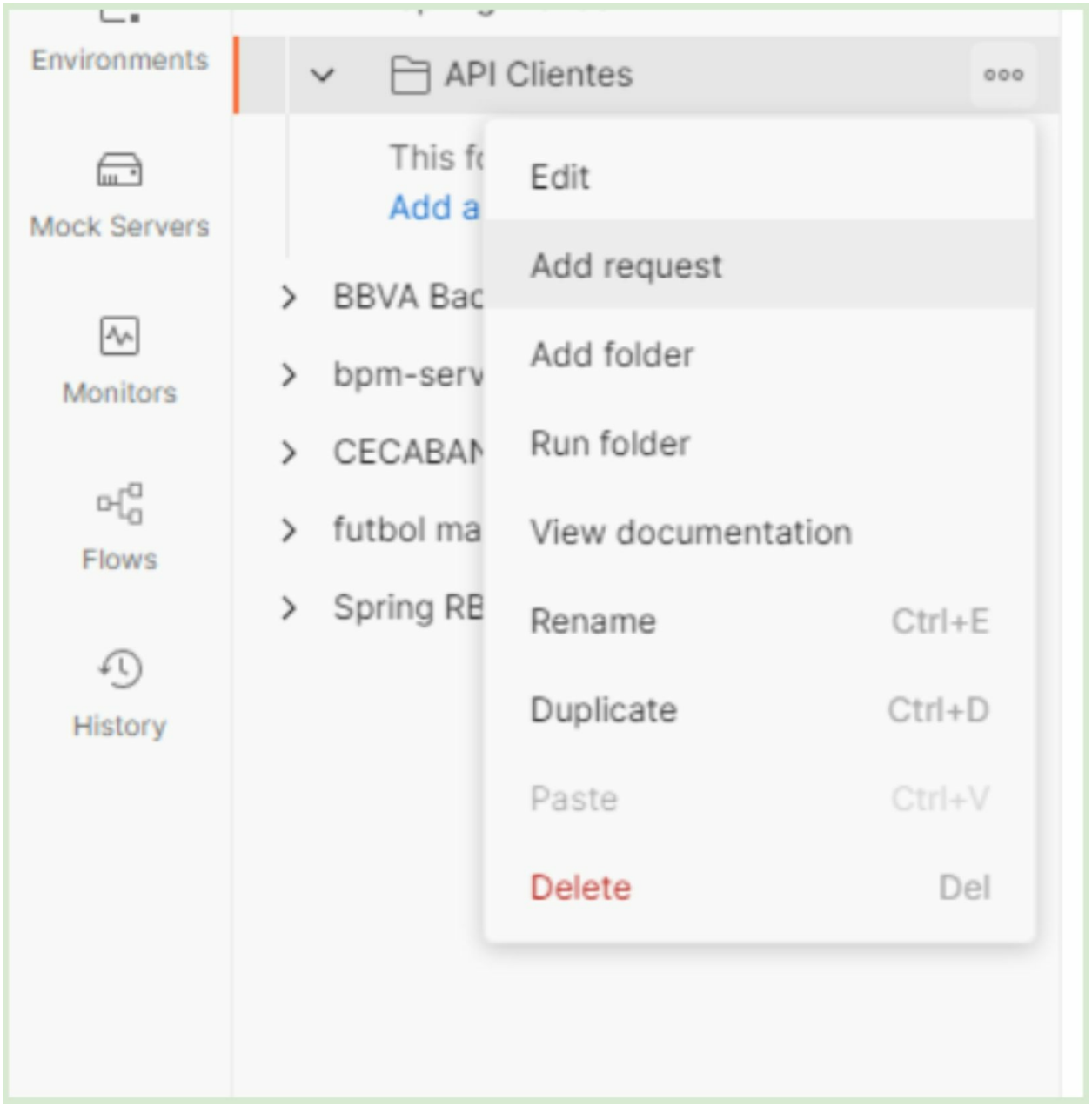
Lo haremos de la misma manera que lo hicimos con la colección. Haremos un click en la carpeta y en el frame central se presenta una carátula de la misma. La cual mediante el icono de “lápiz” ubicado al lado del nombre podremos modificarlo. Renombramos entonces la carpeta con el nombre “API Clientes”



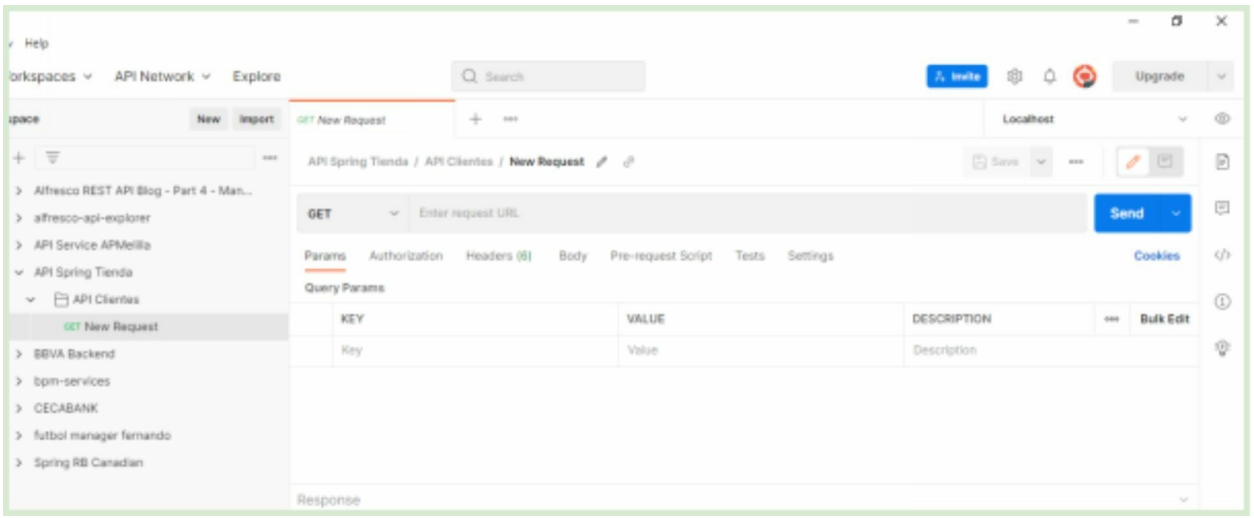
Ahora sí, tenemos nuestra estructura jerárquica (colección/carpeta) lista para incorporar nuestras peticiones REST que atacaran a la API de clientes definidas en Spring. Vamos a crear entonces, tres peticiones. Cada una asociada a los tres endpoints definidos en nuestro controlador cliente.

3.3.3 Consumiendo servicios con Postman

Nos situamos sobre la carpeta “API Clientes” y presionamos el botón derecho del mouse. Se presenta un menú emergente con la opción “Add Request”

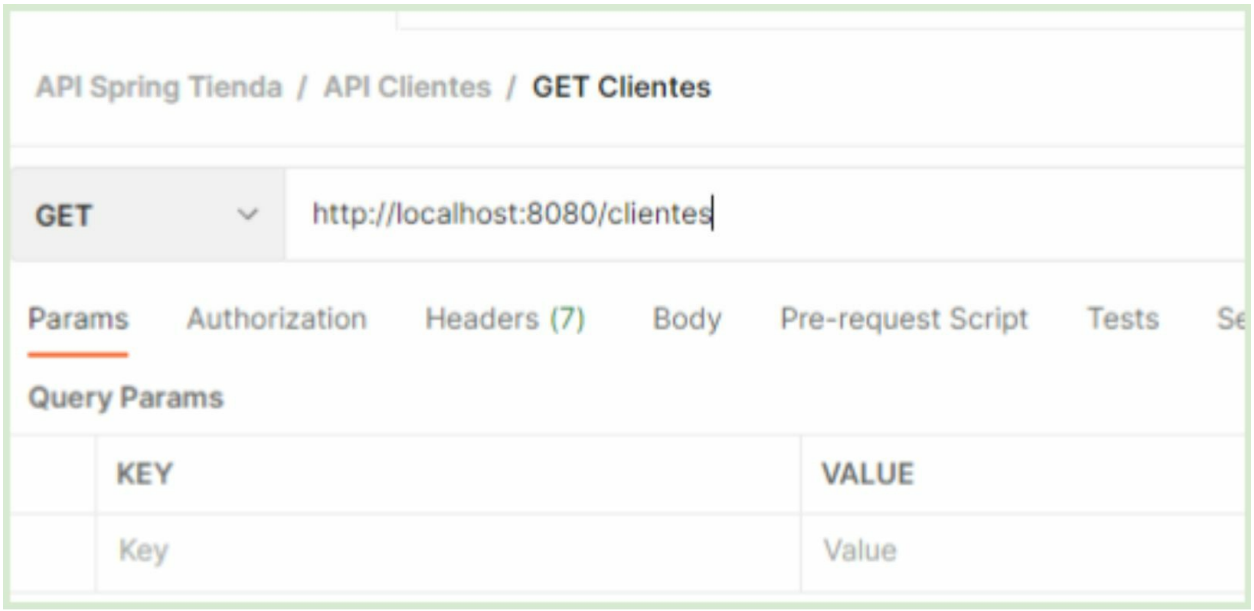


Al hacer click sobre la opción “Add request”, Postman creará una nueva “Petición WEB” también con un nombre por defecto que renombramos de la misma manera que lo hicimos con la colección y la carpeta.



Como pueden observar, se ha creado una petición con el nombre “New Request”. Haremos doble click sobre el nombre de la petición para que se abra en el marco central y así poder cambiar el nombre también haciendo click en el icono de “lápiz”.

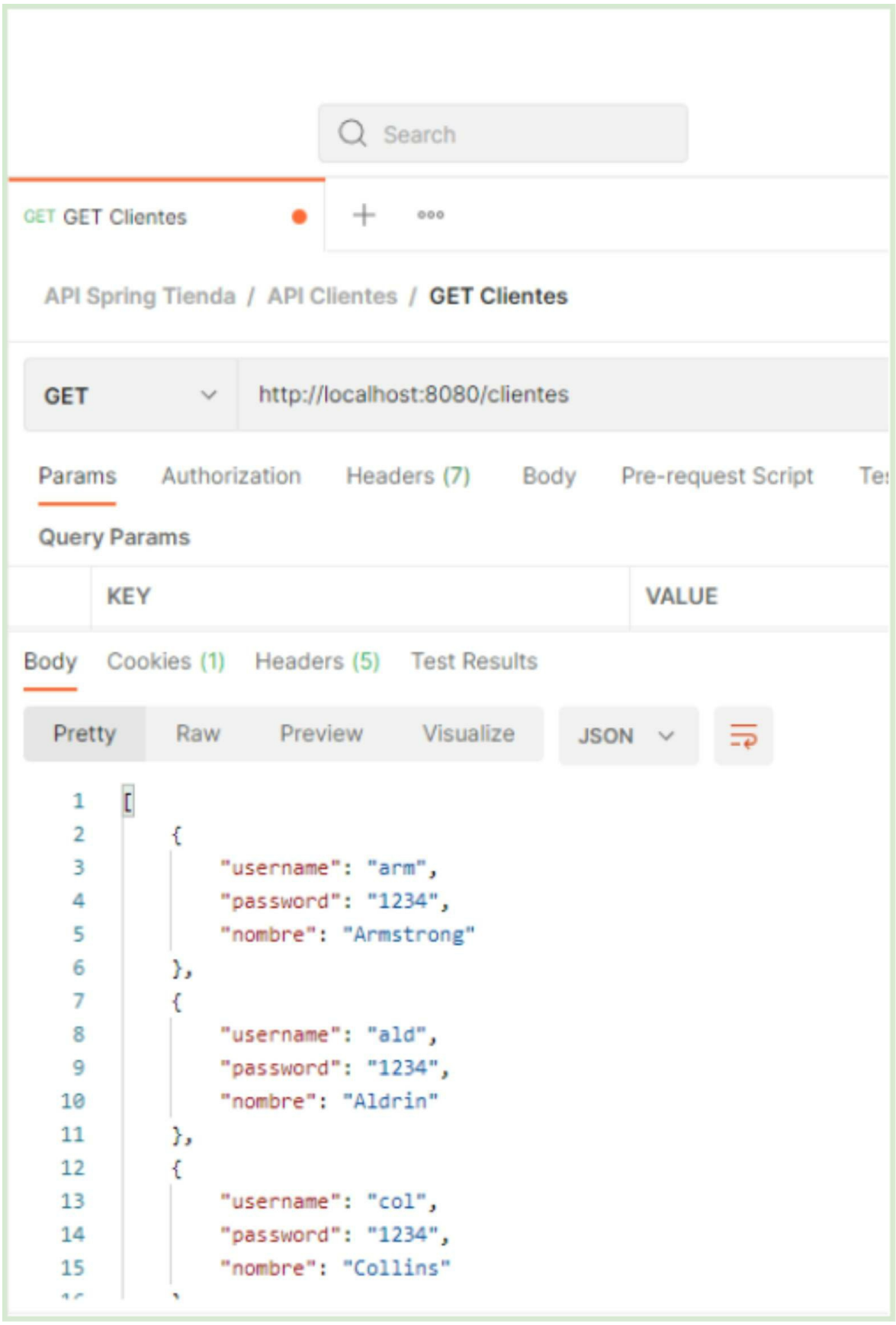
Le asignaremos el nombre “GET Clientes” y en el campo de dirección (a lado del desplegable que muestra la selección “GET”) introduciremos la siguiente dirección web http://localhost:8080/clientes



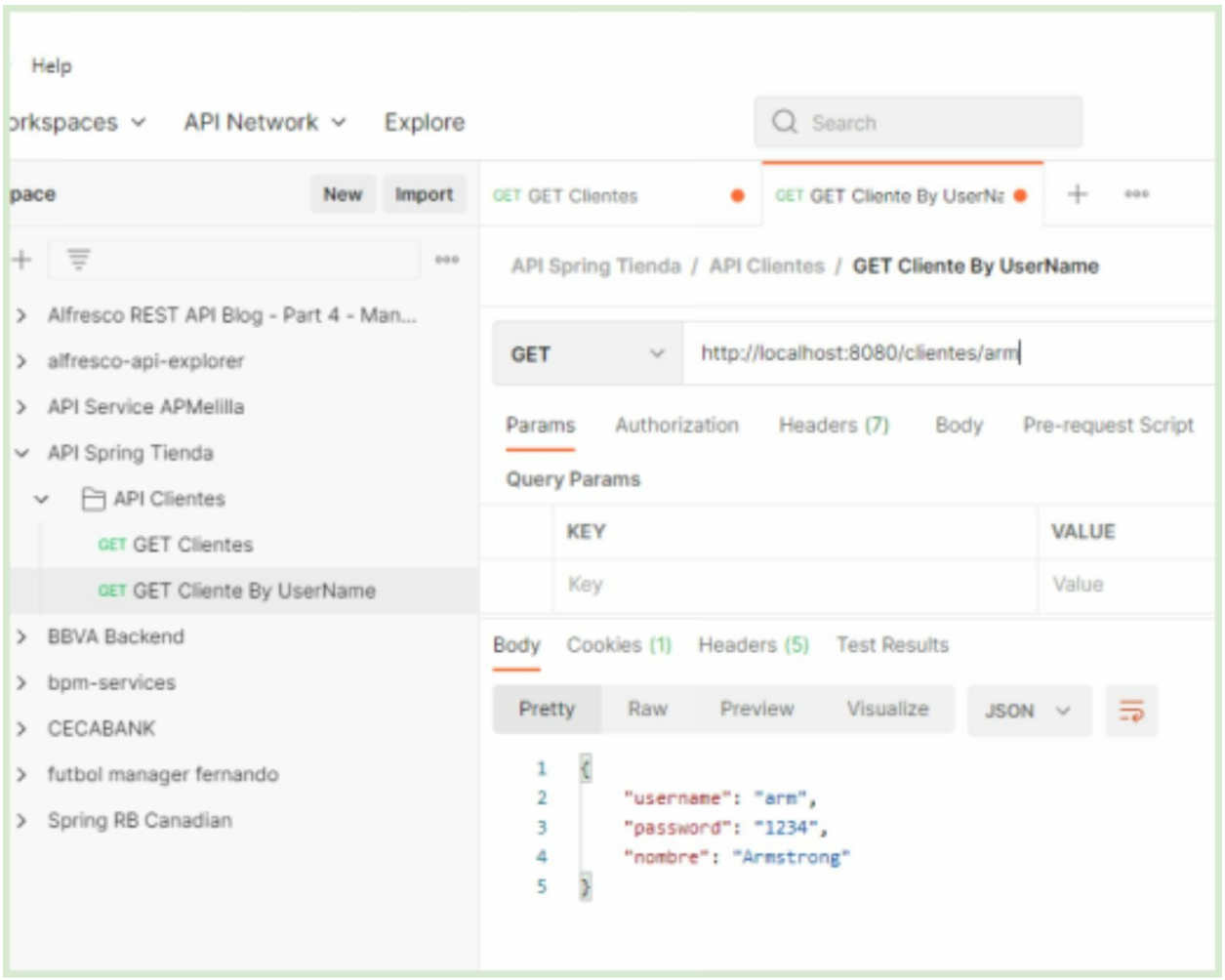
Hemos creado una petición Postman para atacar y consumir el endpoint “clientes” de la API. Ahora sí, podemos finalmente hacer click en el botón “Send” para ejecutar la petición.

Una vez ejecutada la petición se mostrará en el marco inferior la devolución del servicio consumido. En este caso hemos ejecutado el endpoint que devuelve todos los clientes por lo tanto obtendremos como respuesta un array de tipo JSON con el listado de

los mismos.


Search
GET GET Clientes
API Spring Tienda / API Clientes / GET Clientes
GET
http://localhost:8080/clientes
Params
Authorization
Headers (7)
Body
Pre-request Script
Query Params
KEY
VALUE
Body
Cookies (1)
Headers (5)
Test Results
Pretty
Raw
Preview
Visualize
JSON
[
{
"username": "arm",
"password": "1234",
"nombre": "Armstrong"
},
{
"username": "ald",
"password": "1234",
"nombre": "Aldrin"
},
{
"username": "col",
"password": "1234",
"nombre": "Collins"

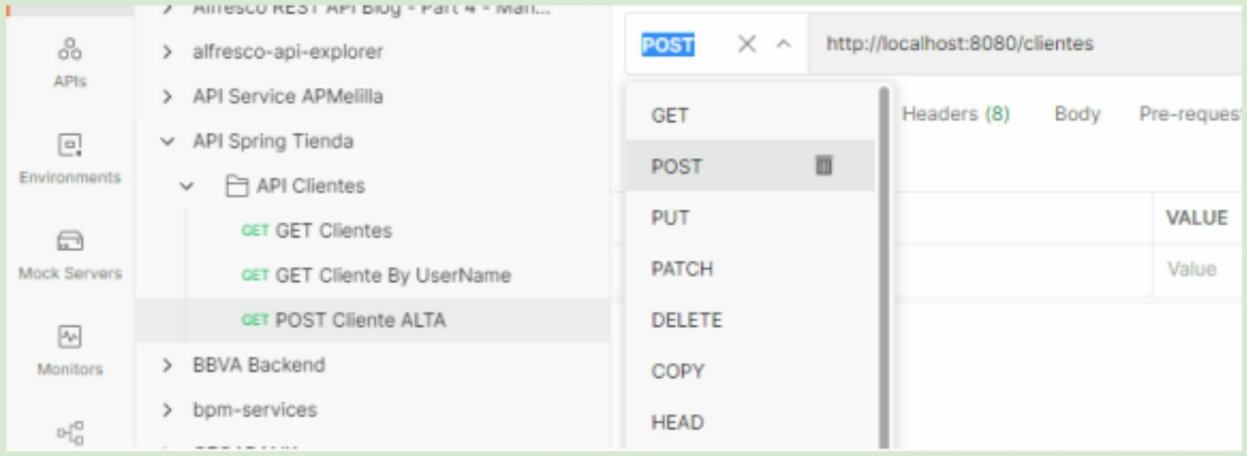
Vamos a crear una nueva petición, en este caso para consumir el endpoint que retorna un cliente por “userName”. Para llevar a cabo esta acción simplemente repetiremos el paso anterior. Sobre la carpeta API Clientes crearemos una nueva petición y la renombramos a “GET Cliente By userName”. Y le agregaremos la url correspondiente (http://localhost:8080/clientes/arm)



Al ejecutar la petición, obtendremos el resultado del usuario en formato JSON como se muestra en el panel inferior de la herramienta.

Si bien todas estas peticiones las podríamos haber realizado con un simple explorador, es muy conveniente contar con un software especializado como POSTMAN que nos permita organizar las llamadas correspondientes a nuestra API. De esta manera, podemos llevar un control de la API a futuro permitiendo pruebas y también realizar ejecuciones más sofisticadas con test unitarios de cada endpoint.

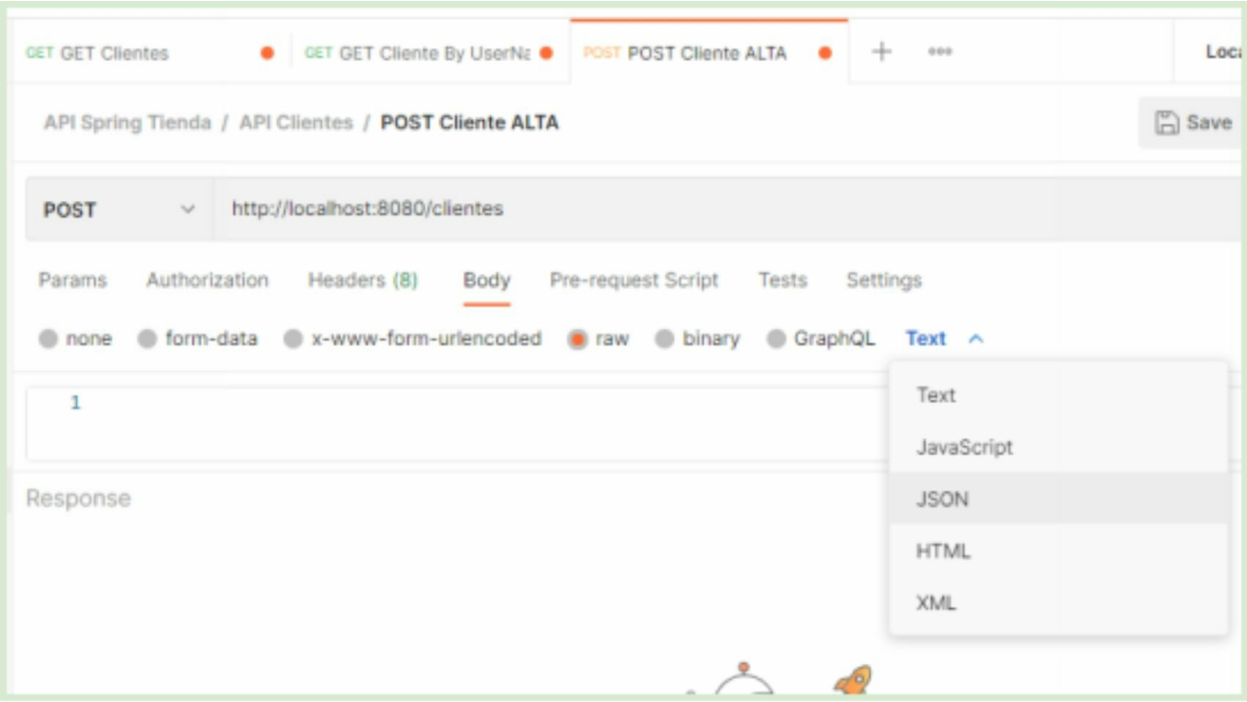
Finalmente crearemos nuestra petición para ejecutar el endpoint de creación de clientes. Repetiremos el paso de creación de una nueva petición. Esta vez con el nombre “POST Cliente ALTA” e ingresamos en la dirección la URL http://localhost:8080/clientes ¡pero atención! Esta vez con un pequeño cambio. En el desplegable que se encuentra al lado de la dirección tendremos que seleccionar la opción POST, ya que el servicio está definido para exponerse mediante este mecanismo.



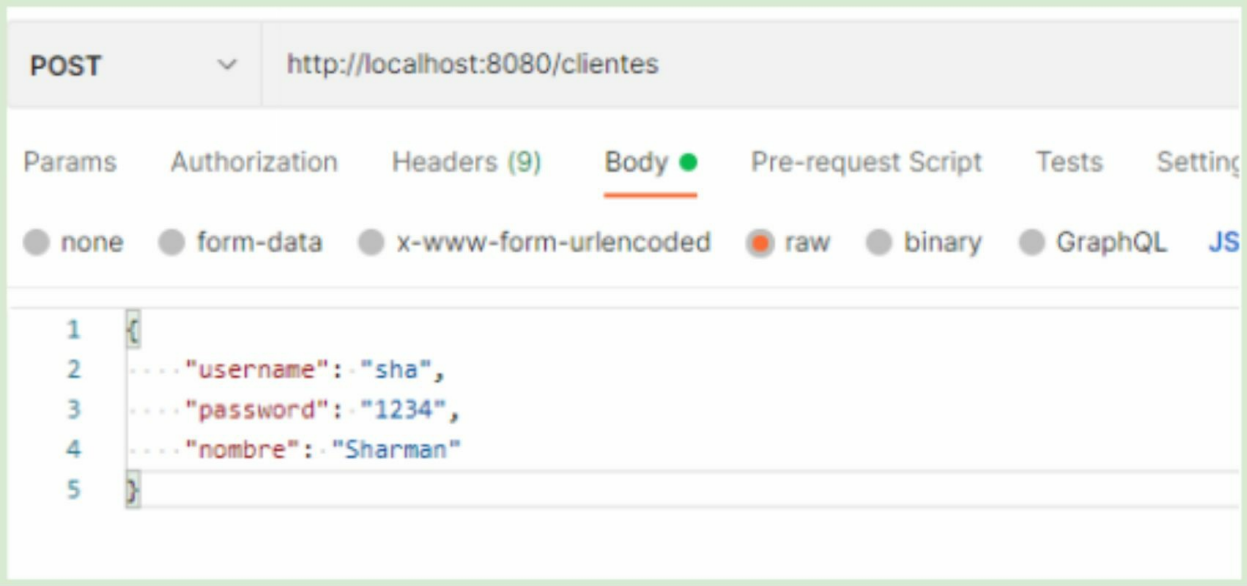
Ya tenemos todo casi listo para el nuevo “ataque”, pero casi, porque nos está faltando algo que es fundamental. Pensemos, este servicio REST nos permite crear un cliente, pero para hacerlo, deberá obtener los datos del mismo como el usuario, nombre y password. Sin esta información nuestro método java en el back-end estará imposibilitado de realizar la creación.

Es entonces que como consumidores del servicio REST, en este caso por medio de la herramienta POSTMAN, debemos suministrar nosotros esa información de cliente. Y lo proveeremos expresado en JSON que es el formato por el cual se “entienden” los servicios REST.

Nos ubicamos en la solapa “body” de la petición y hacemos click en el radio button “raw”. A continuación seleccionamos la opción JSON ya que es el formato en el cual vamos a expresar la información del cliente a crear.

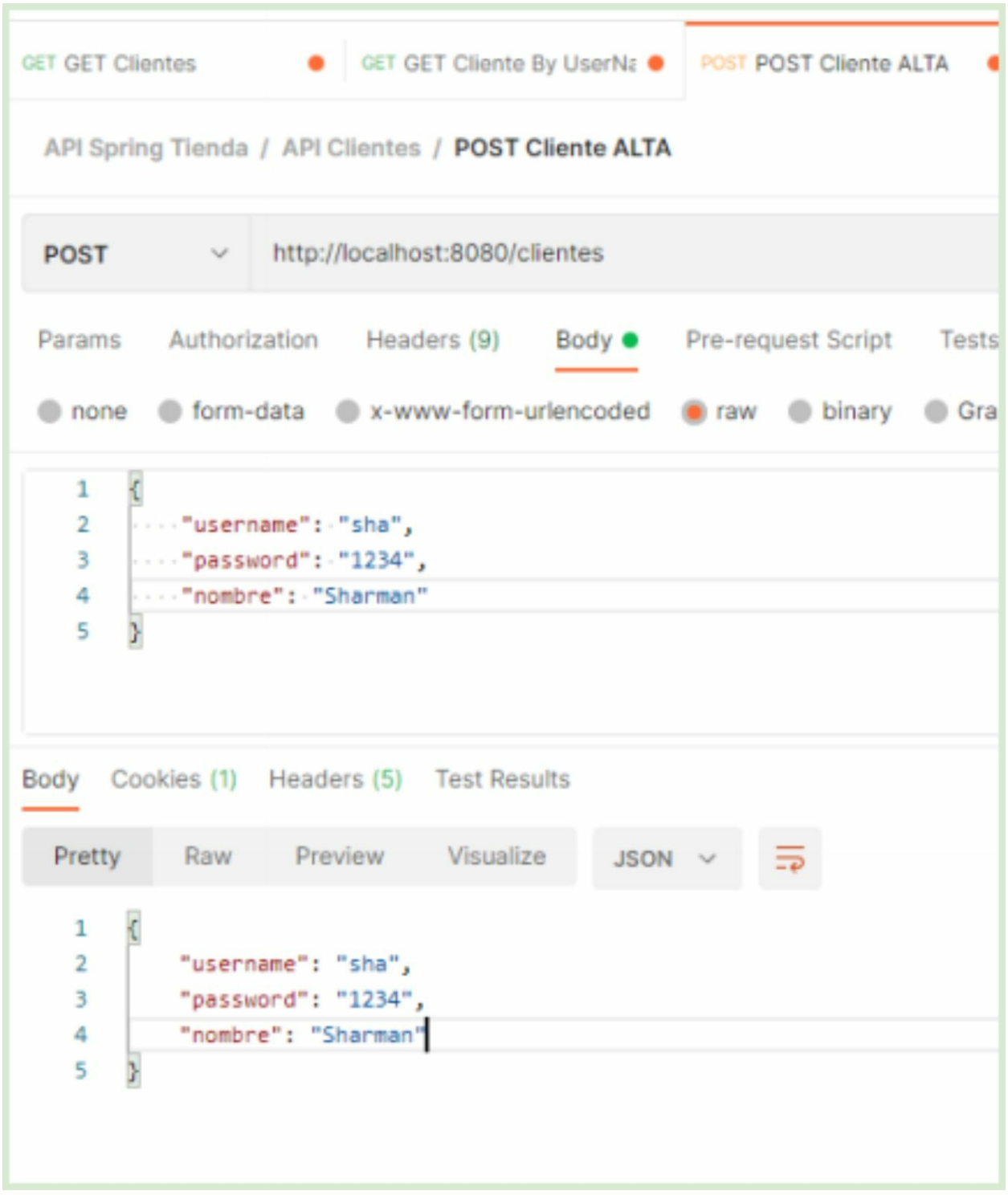


En el campo de texto del body vamos a incorporar la información del cliente en formato JSON como se muestra en la siguiente imagen.

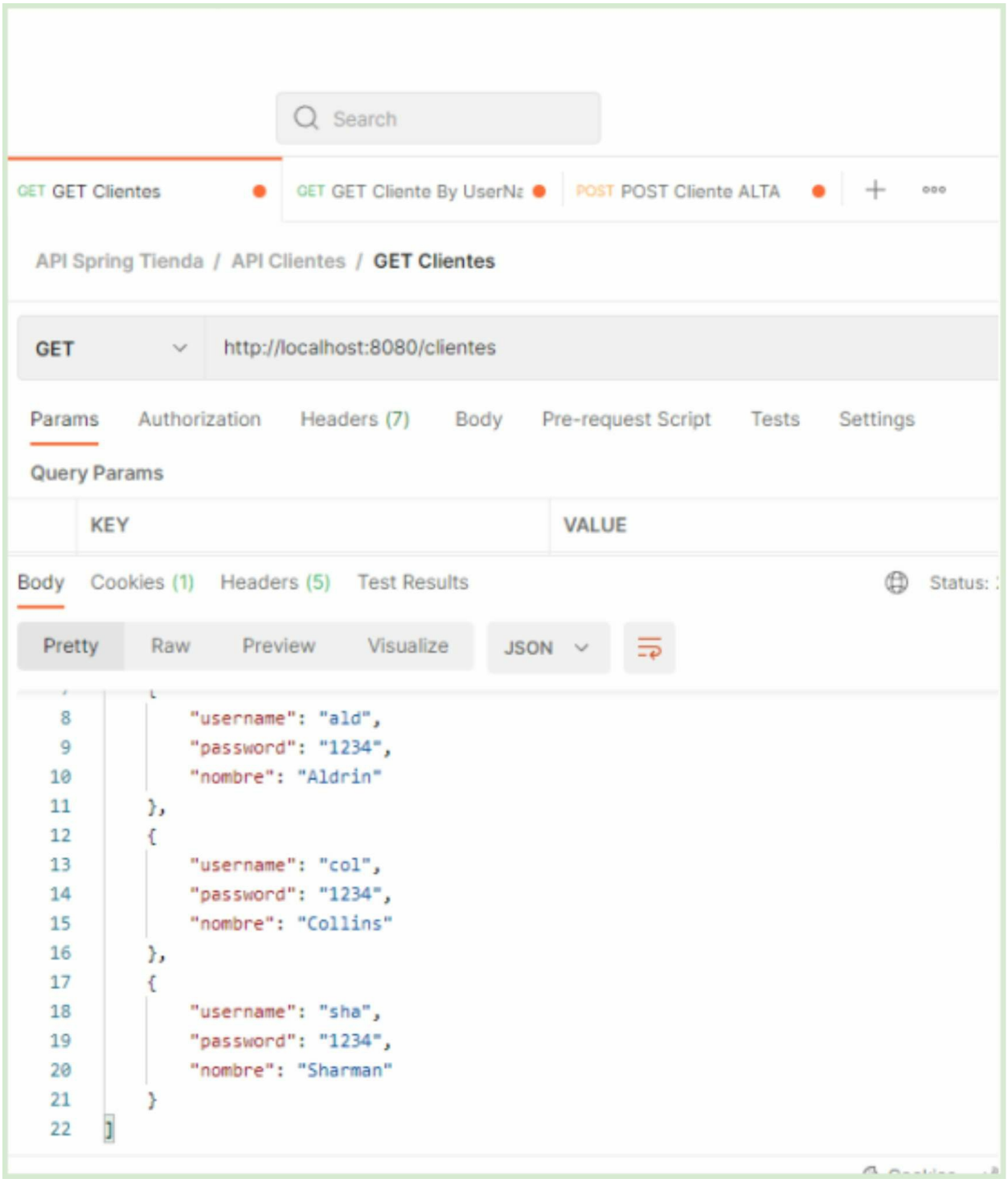


Para obtener el formato exhibido en la imagen anterior deberemos hacer click en el link azul “Beautify” situado debajo del botón “Send” y el link “Cookies”.

Antes de ejecutar la petición recordar reiniciar Spring Boot para que tome la nueva programación del endpoint de creación de clientes. Al ejecutar la nueva petición POST, nuestro back-end adhiere a la lista el nuevo cliente.



Esto significa que se ha creado un nuevo cliente con éxito en el sistema. Para comprobarlo podremos ejecutar nuevamente el servicio que retorna toda la lista de clientes.



Como podemos observar en la última imagen, ahora el servicio de recupero de clientes nos devuelve también el último creado.

3.3.4 Implementando la modificación de un cliente

De acuerdo a las convenciones y buenas prácticas para el diseño de API's REST bien formadas deberemos utilizar el mecanismo HTTP PUT para todo aquel servicio que sea el encargado de modificar o editar algún valor en la información del recurso.

Siguiendo estos lineamientos procederemos a implementar el nuevo método Java para la modificación del cliente.

```
    @PutMapping("/clientes")
    public Cliente modificacionCliente(@RequestBody Cliente cliente){

        return cliente;
    }

}
```

El nuevo método para modificación de clientes presenta prácticamente la misma fisonomía que el método de alta, ya que recibe también un cliente por "Request Body" y retorna ese mismo una vez completada la operación. No obstante, prestemos atención en el pequeño cambio en la anotación que decora el método. Y es que en este caso el mecanismo de acceso es mediante PUT.

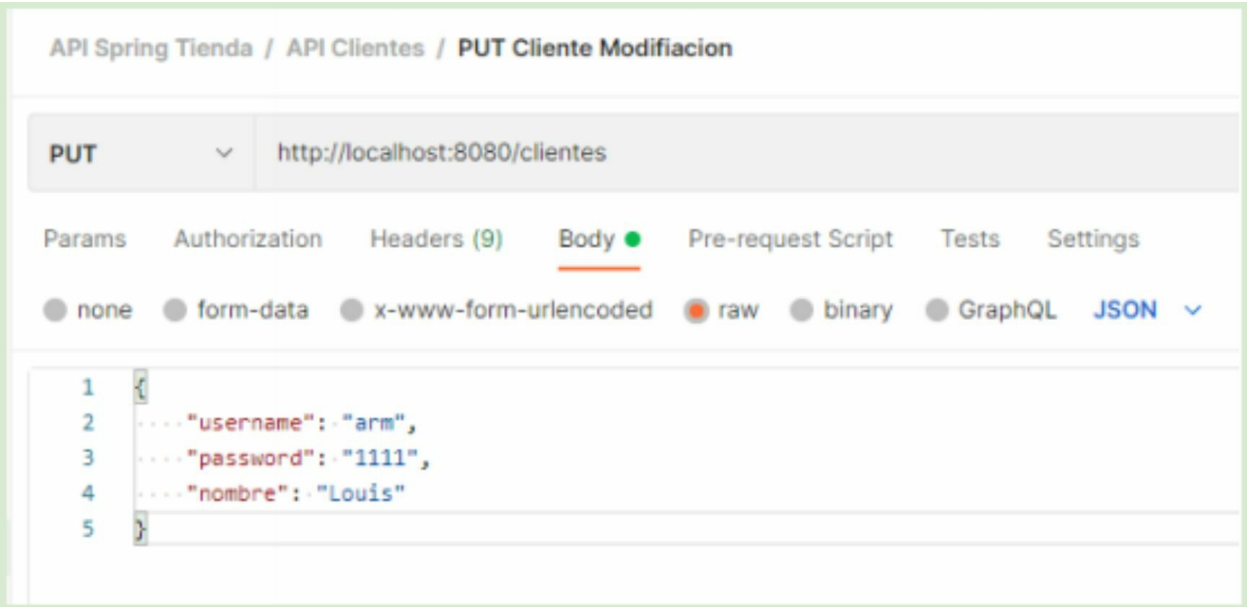
Como los casos anteriores, este endpoint sobrecarga a los anteriores diferenciándose solamente por el método de acceso PUT. Ahora implementemos la lógica de negocio de este método. Para ello, deberemos efectuar dos acciones. La primera será buscar el usuario por el *"userName"* y en el caso de encontrarlo, se procede a implementar el algoritmo para cambiar el valor de los atributos con los nuevos valores que se suministran por "body" en la petición de PUT.

```java
@PutMapping("/clientes")
public Cliente modificacionCliente(@RequestBody Cliente cliente){

    Cliente clienteEncontrado = clientes.stream().
            filter(cli -> cli.getUsername().equalsIgnoreCase(cliente.getUsername())).
            findFirst().orElseThrow();

    clienteEncontrado.setPassword(cliente.getPassword());
    clienteEncontrado.setNombre(cliente.getNombre());

    return clienteEncontrado;
}
```

Como se observa, el primer fragmento de código es similar al método para el recupero de usuario por *"userName"*. "UserName" es el identificador único de los clientes de nuestro sistema, por lo tanto será el único atributo inmutable. Es decir, el único que no se podrá modificar ya que servirá como clave para localizar el cliente que se necesita modificar. En el caso de que ese cliente exista en el sistema, se le "setean" los nuevos valores para los atributos *"password"* y *"nombre"*.

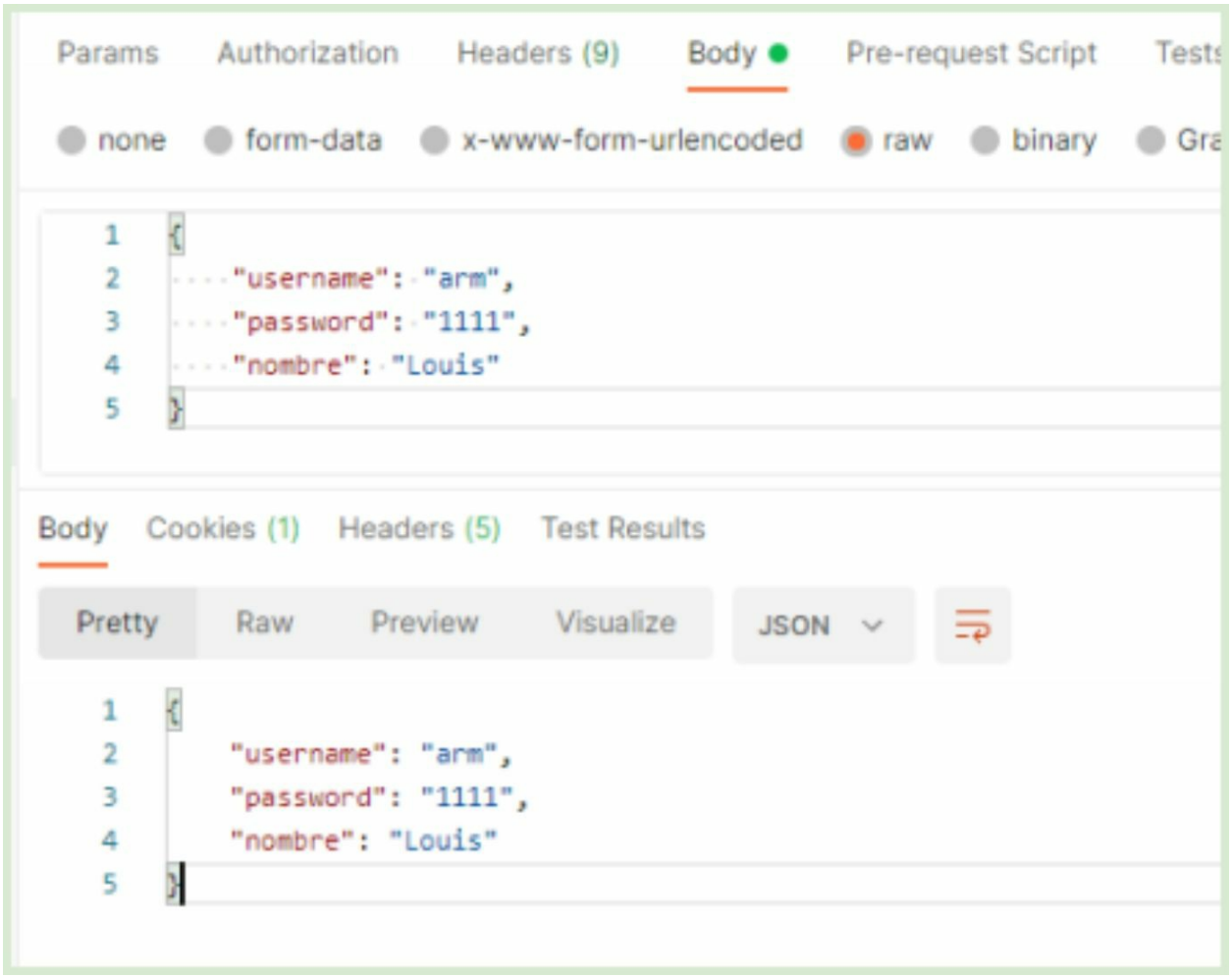
Reiniciamos Spring Boot y abrimos POSTMAN nuevamente para incorporar la nueva petición y ejecutar el servicio de PUT.

Creamos un nuevo "Request" con el nombre "PUT Cliente Modificación" siempre dentro de la carpeta "API Clientes" de la colección "API Spring Tienda". Seleccionamos la opción PUT en el desplegable de metodos y escribimos la dirección adecuada http://localhost:8080/clientes



Como se puede apreciar en la imagen, agregaremos un “Body JSON” a la petición con la información del cliente que deseamos modificar. En este caso modificaremos el cliente con “*userName*” “*arm*” cuyo nombre actual es “Armstrong” con clave “1234” por el nombre “Louis” y la password “1111”.

Ejecutamos la petición y POSTMAN nos presenta la siguiente devolución en el panel inferior de la herramienta



La respuesta es el mismo JSON proporcionado por nosotros en el “Body Request”. Para comprobar que la modificación ha sido efectiva volvemos a ejecutar el servicio que nos devuelve todos los clientes del sistema.

```
[
    {
        "username": "arm",
        "password": "1111",
        "nombre": "Louis"
    },
    {
        "username": "ald",
        "password": "1234",
        "nombre": "Aldrin"
    },
    {
        "username": "col",
        "password": "1234",
        "nombre": "Collins"
```

Como podemos apreciar en la lista retornada, se muestra el usuario con “username” “arm” ahora con los datos de “password” y “nombre” actualizados. Finalmente implementaremos el último método que nos falta. El de eliminación de usuario.

Por convención REST, todo aquel endpoint que realiza una acción de eliminación de recurso debe ser accedido por medio del mecanismo HTTP DELETE.

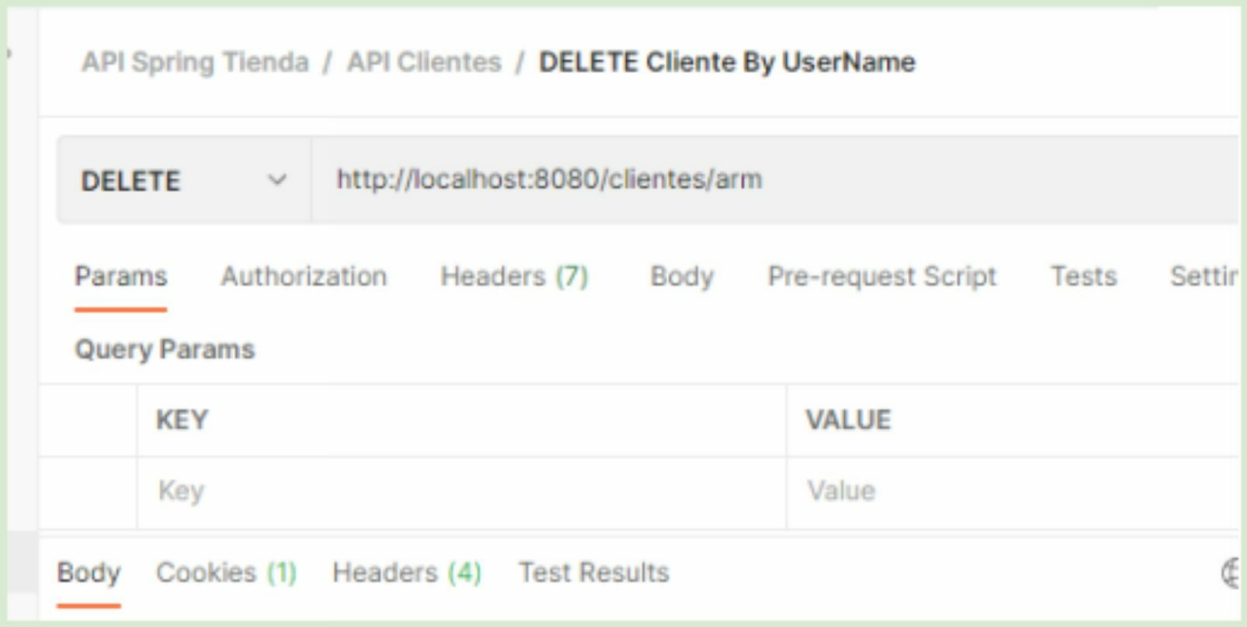
Nuestro método de eliminación presentará este aspecto.

```
    @DeleteMapping("/clientes/{userName}")
    public void deleteCliente(@PathVariable String userName){

        Cliente clienteEncontrado = clientes.stream().
                filter(cli -> cli.getUsername().equalsIgnoreCase(userName)).
                findFirst().orElseThrow();

        clientes.remove(clienteEncontrado);
    }


}
```

Este método es muy parecido en su firma al método de recupero de un cliente por *"userName"*. Es que de hecho, siempre previo al borrado, se deberá localizar el recurso mediante su clave única. De manera tal, que la implementación también es casi idéntica, con la diferencia que una vez encontrado el elemento se remueve de la lista.

Para este caso en particular, nuestro método de borrado no retorna ningún tipo de información. (Más adelante veremos las buenas prácticas utilizando los códigos de respuesta HTTP correspondientes para cada operación).

Comprobar que este endpoint sobrecarga en su firmado a todos los demás. diferenciándose del resto por el medio de acceso, que será "DELETE" para este caso. Reiniciamos Spring y creamos en POSTMAN la petición para este nuevo servicio utilizando el nombre "DELETE Cliente by UserName" agregando la dirección http://localhost:8080/cliente/arm y seleccionando el método "DELETE" en el desplegable correspondiente.



All ejecutar la petición, se borrara el recurso cuyo *"userName"* es "arm". Para comprobarlo, ejecutar el servicio que recupera todos los clientes.

Observar que ahora en la respuesta ya no se presenta el usuario "arm", eliminado en la acción anterior.

3.4 Capa controller CRUD - RequestMapping

En esta sección veremos cómo unificar los nombres de todos los endpoints de un controlador.

3.4.1 Unificación de nombres a nivel controlador

Si observamos con detalle cada método de nuestra clase *"ClientesRestController"*, podremos determinar que todos los endpoints son expuestos utilizando la misma nominación: *"/clientes"*. De acuerdo a las convenciones REST, todos los métodos enfocados a la gestión de un mismo recurso deben mantener el mismo nombre.

Spring dispone de una anotación a nivel de clase que se llama @RequestMapping y permite unificar todos los nombres de los endpoints que contiene. Es por eso que lo vamos utilizar en este caso para centralizar el nombre "clientes" en un único punto de la siguiente manera.

```
@RestController
@RequestMapping("/clientes")
public class ClienteRestController {
```

Observar que la nueva anotación aplica a la clase en general. Por consiguiente, ahora, deberemos remover el nombre del endpoint para cada uno de nuestros métodos. Por ejemplo, el método que recupera todos los clientes ahora presenta este aspecto.

```
@GetMapping
public List<Cliente> getClientes(){

    return clientes;
}
```

Observar con detalle que hemos removido toda la sección de parámetros de la anotación @GetMapping ya que es ahora la anotación @RequestMapping la que contendrá la dirección de

todos los endpoints que la componen.

Podremos entonces realizar la misma limpieza para los demás métodos con excepción de los que reciben como argumento el *"userName"*, que quedarán de la siguiente manera.

```
@DeleteMapping("/{userName}")
public void deleteC1iente(@PathVariable String userName){

    Cliente clienteEncontrado = clientes.stream().
            filter(cli -> cli.getUsername().equalsIgnoreCase(userName)).
            findFirst().orElseThrow();

    clientes.remove(clienteEncontrado);
}
```

Para este caso solo hemos removido el prefijo de la dirección y dejado solo el parámetro que recibe.

Al ejecutar nuevamente las pruebas utilizando POSTMAN podremos verificar que nada ha cambiado desde el "mundo exterior". Sin embargo, en nuestra definición de código hemos logrado la unificación de nombre de API desde un punto único de control, evitando así, cualquier tipo de ambigüedades que podrían conllevar a errores futuros.

3.6 Capa controller CRUD Códigos de respuesta de éxito

En esta sección veremos los códigos de respuesta HTTP que deberán entregar los servicios REST para seguir las convenciones y buenas prácticas.

3.6.1 Códigos de respuesta HTTP

Algo que nunca vemos cuando accedemos mediante un explorador a un sitio como “google”, es que si el acceso a la página fue exitoso, el navegador, de fondo, obtendrá una respuesta numérica por parte del servidor. El número del éxito en el protocolo HTTP es el 200.

Cada vez que un servidor puede entregar de manera exitosa una página, además del código HTML, Javascript y CSS que la componen, entregará una serie de información en las llamadas cabeceras HTTP. Esta información generalmente no es exhibida para el usuario final del navegador (salvo que inspeccionamos las herramientas más sofisticadas).

En dicha cabecera HTTP se almacena el código de respuesta de la petición que en los casos de éxito suele ser el número 200. Es así, que para cumplir con las convenciones y buenas prácticas REST debemos añadir a nuestras respuestas (que en la mayoría de casos serán de tipo JSON) el código de respuesta HTTP que produce el consumo del servicio. Para implementarlo en Spring Boot, debemos apoyarnos en una clase de tipo envoltorio llamada *“ResponseEntity”* (no es una anotación, es una clase). Esta clase deberá estar definida en la sección de retorno situada en la asignatura del método.

Lo implementaremos primero en aquel método que, a priori, no debería presentar problemas o excepciones a la hora de ejecutarse. Estamos hablando del método *“getClientes”*, que recupera todos los clientes del sistema. A continuación modificaremos el método de la siguiente manera.

```
@GetMapping
public ResponseEntity<List<Cliente>> getClientes(){

    return ResponseEntity.ok(clientes);
}
```

Observar con detalle los cambios realizados. En primer lugar, hemos definido que este método, ya no retorna una lista de clientes, sino que en este caso, devolverá un objeto del tipo *“ResponseEntity”* que tendrá información de respuesta, pero además, contendrá la lista de clientes.

Gracias a este objeto, podremos no solo devolver el contenido producido por este servicio sino también, información adicional como el código de respuesta HTTP. Es por eso que se trata de una clase de tipo “Wrapper” (envoltorio) ya que embebe un objeto dentro de ella.

En el cuerpo del método, podemos observar que no se retorna directamente la lista de clientes, sino que se crea un objeto *"ResponseEntity"* mediante una factoría estática y propia mediante el método *"ok"*. Este método *"ok"* se corresponde directamente con un valor numérico de código de respuesta HTTP que para este caso es 200.

A continuación se muestra una tabla con los códigos de respuesta esperados para cada tipo de servicio de acuerdo a las convenciones y buenas prácticas REST.

Verbo	***Código de respuesta esperado de éxito***
GET	**200**
POST	**201**
PUT	**200,204**
DELETE	**200,204**

Como se puede ver en la tabla, la mayoría de verbos HTTP deben retornar un código de respuesta 200 cuando la acción es completada con éxito. Todos, excepto en el caso de POST que retorna un 201 (Creación de recurso).

En el caso de PUT y DELETE pueden retornar un 200 o 204 para el caso de que la acción se complete, pero no retorna ningún recurso como respuesta (No content).

3.6.1 ResponseEntity

Retomando nuestra clase de controlador de clientes, haremos un pequeño cambio que no afectará la funcionalidad del programa. Nuestro método para recuperar todos los clientes tenía el siguiente aspecto.

```
@GetMapping
public ResponseEntity<List<Cliente>> getClientes(){

    return ResponseEntity.ok(clientes);
}
```

Se puede observar que *"ResponseEntity"* envuelve una lista de clientes. Este tipo de dato declarado dentro del segmento genérico puede modificarse para ser aún más flexible. Vamos entonces a cambiar el tipo de dato retornado por el "wildcard" <?>.

```
@GetMapping
public ResponseEntity<?> getClientes(){

    return ResponseEntity.ok(clientes);
}
```

Este "wildcard" permite mayor flexibilidad ante un cambio en el cuerpo del método, sobre todo que ya no debemos preocuparnos por definir que tipo de dato se retorna. Por ejemplo, puede que el día de mañana nuestra Lista de clientes se transforme en *"Map"* o *"Set"*, y por lo tanto, este cambio no afectará la firma del método.

Ahora vamos a realizar la misma modificación para todos los métodos que retornan 200, que son GET y PUT. Aquí la modificación del servicio *"getCliente"*.

```java
@GetMapping("/{userName}")
public ResponseEntity<?> getCliente(@PathVariable String userName){
    return ResponseEntity.ok(clientes.stream().
            filter(cliente -> cliente.getUsername().equalsIgnoreCase(userName)).
            findFirst().orElseThrow());
}
```

En cuanto al servicio de modificación, deberá ser implementado de la siguiente manera.

```java
@PutMapping
public  ResponseEntity<?> modificacionCliente(@RequestBody Cliente cliente){

    Cliente clienteEncontrado = clientes.stream().
            filter(cli -> cli.getUsername().equalsIgnoreCase(cliente.getUsername())).
            findFirst().orElseThrow();

    clienteEncontrado.setPassword(cliente.getPassword());
    clienteEncontrado.setNombre(cliente.getNombre());

    return ResponseEntity.ok(clienteEncontrado);
}
```

La acción de DELETE es un caso especial, ya que en esta ocasión, luego de completar la operación no retorna ningún recurso. Es por eso que utilizaremos la respuesta 204, que significa éxito y “NO CONTENT” al mismo tiempo. Nuestro método entonces debería ser modificado de la siguiente forma.

```java
@DeleteMapping("/{userName}")
public ResponseEntity deleteCliente(@PathVariable String userName){

    Cliente clienteEncontrado = clientes.stream().
            filter(cli -> cli.getUsername().equalsIgnoreCase(userName)).
            findFirst().orElseThrow();

    clientes.remove(clienteEncontrado);

    return ResponseEntity.noContent().build();
}
```

Por último vamos a implementar el caso de POST, que a diferencia de la mayoría deberá retornar un código de respuesta 201 (CREATED).

Siguiendo las recomendaciones de la tesis REST, los servicios POST de creación deberían retornar el nuevo recurso creado, pero además se propone también, retornar la URL para acceder a ese nuevo recurso. Es un link que apunta al servicio GET “cliente by userName”.

```java
@PostMapping
public ResponseEntity<?> altaCliente(@RequestBody Cliente cliente){
    clientes.add(cliente);

    //Obteniendo URL de servicio.
    URI location = ServletUriComponentsBuilder
            .fromCurrentRequest()
            .path("/{userName}")
            .buildAndExpand(cliente.getUsername())
            .toUri();


    return ResponseEntity.created(location).body(cliente);
}
```

Forma de enviar la url del endpoint

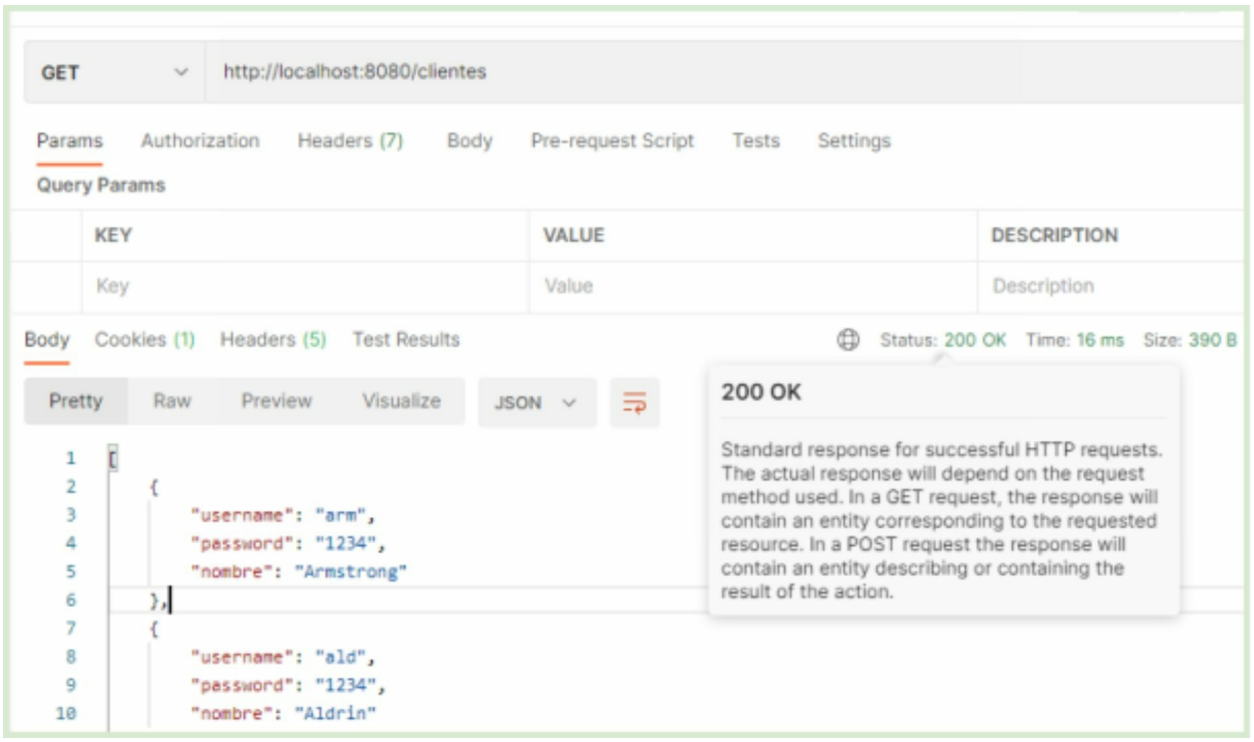
Comentaremos con detalle los nuevos cambios aplicados al método de alta de cliente. En principio, al igual que los demás métodos, hemos declarado como retorno un *"ResponseEntity"* que nos permitirá devolver el código 201 ("CREATED") como se especifica en la última línea. También, con el método *"body"*, se retorna el recurso creado entero como respuesta de la petición.

Vamos a centrarnos en el fragmento de código al estilo DSL que recupera el path actual de este servicio. Como formalidad, REST propone que cada vez que se crea un recurso, se retorne una respuesta en el HEADER con su ubicación para una futura obtención.

Entonces el método *"ServletUriComponentsBuilder.fromCurrentRequest()"* retorna la locación actual del servicio corriente que es http://localhost:8080/clientes. Los métodos subsiguientes *"path"* y *"buildAndExpand"* agregan a dicha URL el sufijo con el path variable *"userName"*. Conformando la URL final http://localhost:8080/clientes/arm y útil para quien consuma el servicio.

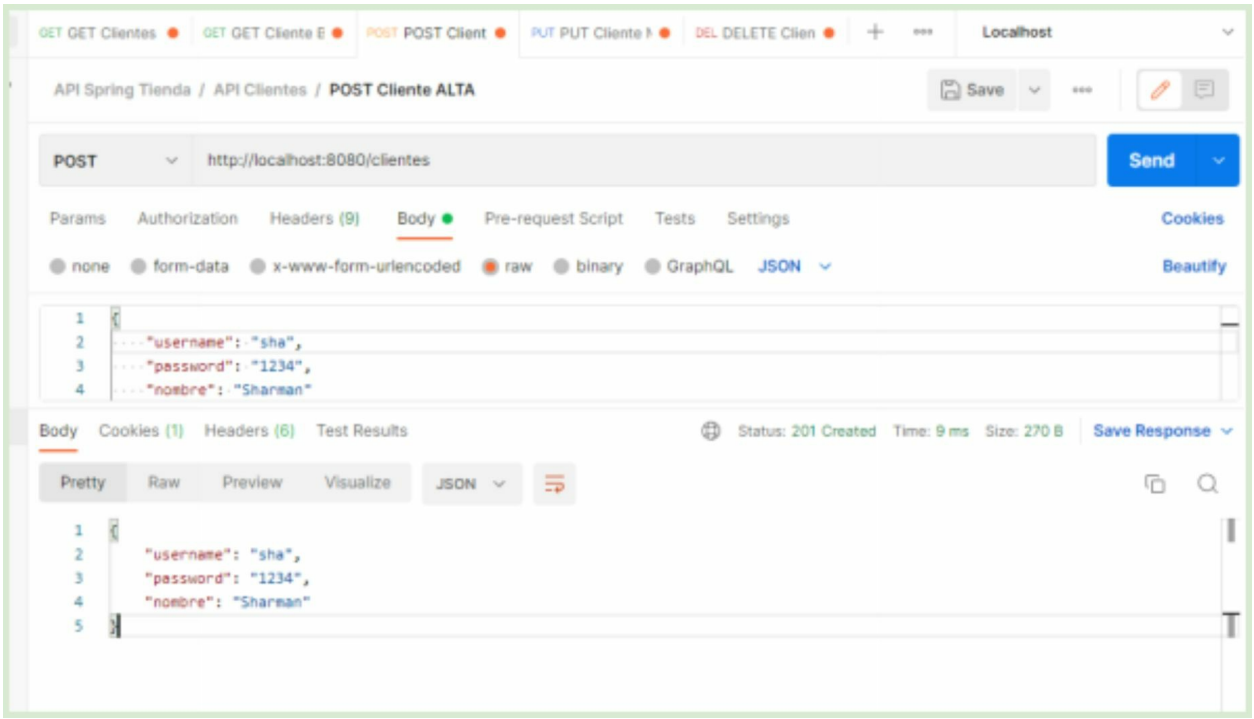
3.6.2 Ejecutando los servicios con POSTMAN

Ahora vamos a probar los servicios nuevamente con POSTMAN para verificar la devolución de estos códigos de respuesta HTTP que hemos implementado en nuestro controlador. Probaremos primero el servicio que retorna a todos los clientes. GET http://localhost:8080/clientes



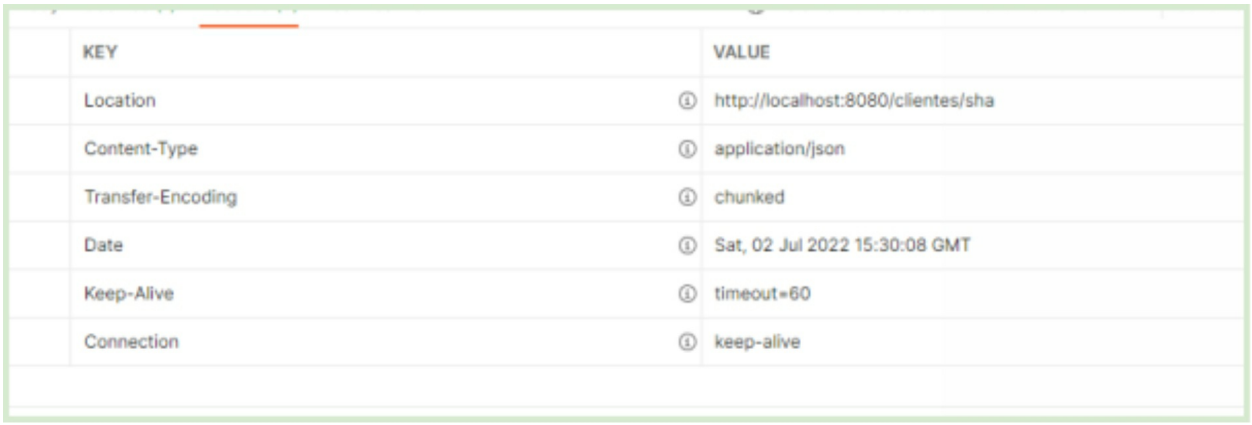
POSTMAN nos informa que la ejecución del servicio ha retornado un código de STATUS 200 OK.

Ejecutamos el servicio POST http://localhost:8080/clientes



Como podemos apreciar, esta ejecución ha retornado un código 201. Ahora vamos a

hacer click en la solapa inferior “Headers” de POSTMAN para verificar las cabeceras retornadas en la respuesta desde el servidor.



KEY	VALUE
Location	http://localhost:8080/clientes/sha
Content-Type	application/json
Transfer-Encoding	chunked
Date	Sat, 02 Jul 2022 15:30:08 GMT
Keep-Alive	timeout=60
Connection	keep-alive

Entre los datos de cabecera se ha retornado el dato “Location” con la URL para recuperar el nuevo recurso creado. Esta URL apunta al servicio que recupera un cliente por “userName” (GET http://localhost:8080/clientes/sha)

Esto valor agregado, hace al servicio un tanto más “hospitalario” con aquellos clientes que lo consuman. Imaginemos una aplicación frontal Angular o React que disponga de un formulario de creación de clientes. Una vez que el usuario completa el formulario, un componente de angular ejecutará este servicio POST para enviar los datos requeridos y crear el recurso en el servidor. Luego de ejecutar la petición se retornara el header con el dato “Location” útil para que este frontal nos haga una redirección a otra vista donde se muestran los datos completos del cliente.

3.5 Capa controller CRUD - Códigos de error

En esta sección veremos los códigos de respuesta de error HTTP que deberán entregar los servicios REST para seguir las convenciones y buenas prácticas en caso de que se produzcan excepciones en la ejecución.

3.5.1 Códigos de respuesta HTTP de error

¡En la vida no son todos éxitos! Es por eso que los servicios deben estar preparados para manejar ciertas excepciones a la regla.

Por ejemplo, el servicio que busca un cliente por *"userName"* no siempre va a entregar resultados. Es decir, no siempre obtendrá ocurrencias para dicha búsqueda. Por ejemplo, si ejecutamos el servicio de la siguiente manera http://localhost:8080/clientes/pepe y de acuerdo a nuestro almacén de clientes predefinido en el controlador, para este caso de búsqueda , el servicio no arroja ningún resultado.

Es en este caso, que el back-end no debería retornar un código de respuesta 200 OK, sino, un 404 ("Recurso no encontrado"). Este código 404 es el mismo que se nos muestra habitualmente en el navegador cuando queremos acceder a una dirección web inexistente. Para el caso, el significado es el mismo: No se encuentra el recurso para satisfacer la búsqueda.

Es importante aclarar que este comportamiento no se trata de un error como tal, ya que el servicio se ejecuta acorde a lo preestablecido pero con la simple excepcion de que no puede satisfacer la búsqueda del cliente solicitado.

Dada esta introducción, vamos a implementar esta respuesta en el servicio de búsqueda de clientes. Primero lo vamos a implementar en la más básica y rústica de las formas para que se logre entender con exactitud el código Java para la devolución del 404. Y una vez implementado y probado de esta manera nos introduciremos en la implementación de la solución más sofisticada.

```java
@GetMapping("/{userName}")
public ResponseEntity<?> getCliente(@PathVariable String userName){

    for (Cliente cliente: clientes) {
        if (cliente.getUsername().equalsIgnoreCase(userName)){
            //En este caso, el usuario fue enccontrado y
            // retornamos el codigo 200 OK con el cliente en el body de respuesta.
            return ResponseEntity.ok(cliente);
        }
    }

    //Si la ejecucion llega a este segmento, es porque no encontro ningun usuario.
    //En este caso construimos una respuesta con el codigo NOT FOUND 404.
    return ResponseEntity.notFound().build();

}
```

Hemos modificado el anterior código de este método a la manera más usual y básica de Java, recorriendo el array de clientes con un "*foreach"*, y preguntando por su nombre de usuario. En el caso de que se encuentre alguna ocurrencia, el código entrará en el "*if*" y responderá con el código de éxito 200 OK. De lo contrario la ejecución seguirá y finalmente responderá con el código NOT FOUND 404.

Hemos retornado a este tipo de implementación más simple para no tener que lidiar con el formato lambda y la introducción de este manejo de excepciones al mismo tiempo, ya que esto puede incurrir en un código un tanto engorroso para absorber en un primer acercamiento.

Ahora si, ya comprendido este manejo. Vamos a implementarlo de una manera más profesional. Para esto, antes que nada vamos a crear una clase de excepción de tipo general que se lanzará cuando un recurso no es encontrado.

3.5.2 Crear excepción de recurso no encontrado

Vamos a crear primero un paquete de excepciones con el nombre *"exceptions"* en *"edu.tienda.core.exceptions"*. Dentro de este paquete crearemos la nueva excepcion con el nombre *"ResourceNotFoundException"* de la siguiente manera.

```
package edu.tienda.core.exceptions;
public class ResourceNotFoundException extends RuntimeException  {

    1 usage
    public ResourceNotFoundException(String message){
        super(message);
    }

    @Override
    public String getMessage() {
        return super.getMessage();
    }
}
```

Hasta aquí es una excepcion común y corriente sin ningún tipo de agregado Spring. Vamos a decorarla con dos anotaciones que nos permitirán aportar un comportamiento más dirigido en cuanto a su propósito y orientado al manejo de servicios REST.

```java
package edu.tienda.core.exceptions;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;

2 usages
@ResponseBody
@ResponseStatus(HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException  {

    1 usage
    public ResourceNotFoundException(String message){
        super(message);
    }

    @Override
    public String getMessage() {
        return super.getMessage();
    }
}
```

La primera anotación @ResponseBody indica que esta excepción, al lanzarse desde algún controlador, formará parte del cuerpo de respuesta de la petición.

El segundo anotador indicará que al lanzarse esta excepcion, se reportara con un código de respuesta 404. Esto significa que Spring creará una *"ResponseEntity"* por nosotros al detectar el lanzamiento de esta excepción. Ahora cambiaremos nuestro código del controlador de manera sutil.

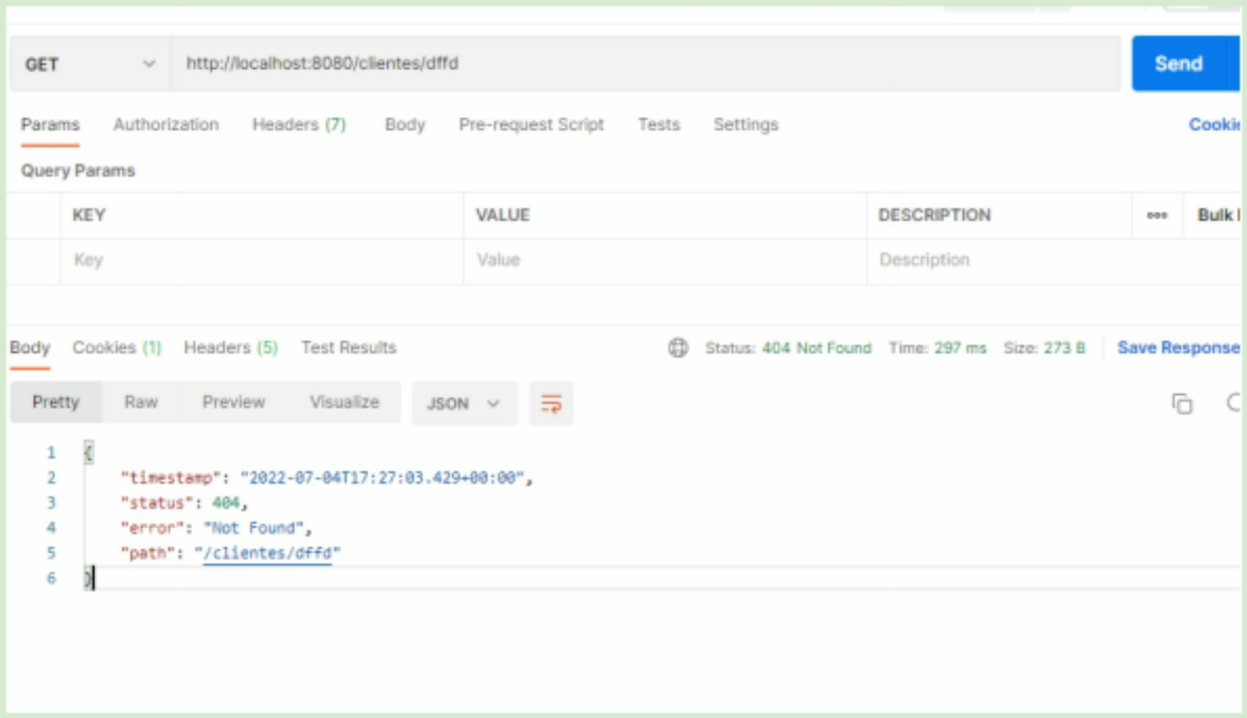
```java
@GetMapping("/{userName}")
public ResponseEntity<?> getCliente(@PathVariable String userName){

    for (Cliente cliente: clientes) {
        if (cliente.getUsername().equalsIgnoreCase(userName)){
            return ResponseEntity.ok(cliente);
        }
    }

    throw new ResourceNotFoundException("Cliente no encontrado");
}
```

Como se puede observar, hemos cambiado la última línea. Ahora simplemente lanzamos la excepcion *"ResourceNotFoundException"* de manera explícita en lugar de construir nosotros el código NOT FOUND con *"ResponseEntity"*.

En este caso, Spring Boot generará ese código de respuesta por nosotros. Spring se encargará de generar el *"ResponseEntity.notFound().build()"* cuando detecte en tiempo de ejecución que se ha lanzado la excepcion *"ResourceNotFoundException"*. Este comportamiento, se logra gracias al decorador @ResponseStatus situado en la parte superior de la clase *"ResourceNotFoundException"*.

Esta modalidad de implementar el servicio es un poco más flexible ya que podremos codificar nuestro controlador orientado a excepciones, delegando a Spring Boot el trabajo de resolver los códigos de respuesta de acuerdo a las anotaciones @ResponseStatus.

Vemos la ejecución desde POSTMAN.



Es importante que observemos el código devuelto “404 Not Found” que se muestra en POSTMAN.

Finalmente desarrollaremos una implementación más sofisticada del método de búsqueda de clientes, que es totalmente equivalente al anterior pero en este caso está basado en el uso de lambdas, opcionales y mapeos.

```
@GetMapping("/{userName}")
public ResponseEntity<?> getCliente(@PathVariable String userName){
    return clientes.stream().
            filter(cliente -> cliente.getUsername().equalsIgnoreCase(userName)) Stream<Cliente>
            .findFirst() Optional<Cliente>
            .map(ResponseEntity::ok) Optional<ResponseEntity<Cliente>>
            .orElseThrow(() -> new ResourceNotFoundException("Cliente " + userName + " no encontrado."));
}
```

Es entendible que el código presentado no es lo más intuitivo a simple vista, y sobre todo, si no tenemos práctica con este tipo de funciones. Pero sí hay que admitir que nuestro código es super conciso, dirigido y en un solo “cartucho” resolvemos un problema sin recurrir a ningún tipo de estructura de control como *“for”* o *“if”*.

En primer lugar se convierte la lista de clientes a tipo *“stream”* dándonos la posibilidad de ejecutar algunos métodos útiles. En este caso utilizamos *“filter”* que es el que necesitamos para realizar un filtrado por *“userName”*. En segundo término se ejecuta el método *“findFirst()”* que retornará un opcional de cliente. El método *“map”* transformara el *“Optional<Cliente>”* en un *“Optional<ResponseEntity<Cliente>>”* y retornará este objeto directamente.

Esto equivale a la siguiente codificación: *“return ResponseEntity.ok(cliente)”*. En el caso de que el opcional de cliente no contenga un objeto, se ejecutará la última línea que provocará la excepción. Luego será Spring el encargado de transformar la respuesta del servicio como NOT FOUND.

3.5.3 Otros tipos de errores

Ahora vamos analizar algún otro tipo de excepción en nuestro servicio. Este tipo de excepcion no es tan natural como el 404, sino que tiene que ver con la forma del uso y consumo de la petición.

Supongamos que determinamos arbitrariamente que el *"userName"* siempre será válido cuando contenga exactamente tres caracteres. Por ejemplo, "arm", "ald" o "col". De acuerdo a esta validación, implementaremos una solución que lanza una excepcion cuando este usuario está mal formado.

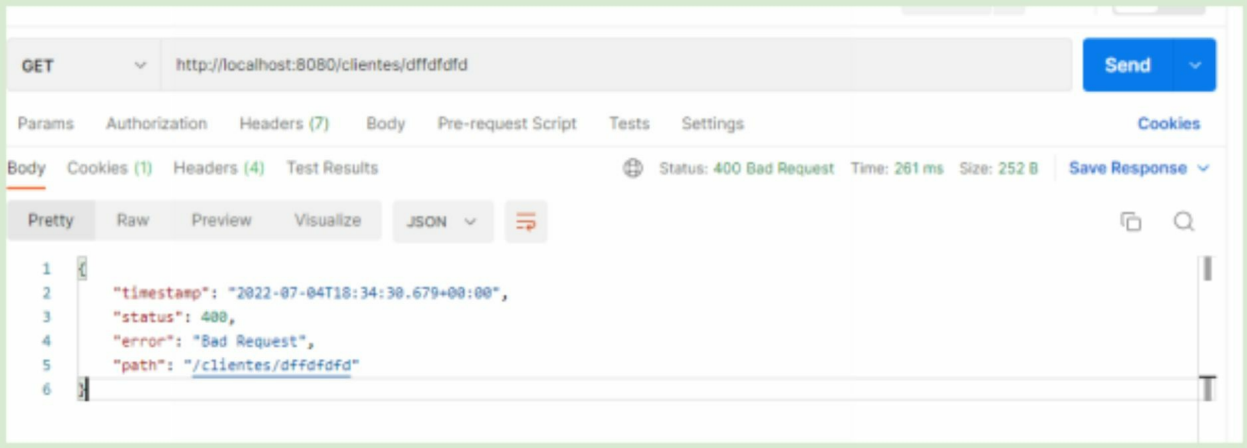
Este tipo de excepción está contemplado en los códigos HTTP como error 400, que significa "BAD REQUEST", es decir que la petición está mal formada por el cliente que consume el servicio. Vamos a crear entonces la excepción con el nombre *"BadRequestException"*, por supuesto, dentro del paquete *"exceptions"*.

```java
package edu.tienda.core.exceptions;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;

3 usages
@ResponseBody
@ResponseStatus(HttpStatus.BAD_REQUEST)
public class BadRequestException extends RuntimeException{

    1 usage
    public BadRequestException(String message){
        super(message);
    }

    @Override
    public String getMessage() {
        return super.getMessage();
    }
}
```

Lo más destacable aquí, nuevamente es la anotación @ResponseStatus que indica que cuando sea lanzada la excepción, Spring retorne un *"ResponseEntity"* con status 400. Vamos a implementar la validación en el servicio de búsqueda.

```java
@GetMapping("/{userName}")
public ResponseEntity<?> getCliente(@PathVariable String userName){

    if (userName.length()!=3){
        throw new BadRequestException("El parametro nombre de usuario debe contener 3 caracteres");
    }

    return clientes.stream().
            filter(cliente -> cliente.getUsername().equalsIgnoreCase(userName)) Stream<Cliente>
            .findFirst() Optional<Cliente>
            .map(ResponseEntity::ok) Optional<ResponseEntity<Cliente>>
            .orElseThrow(() -> new ResourceNotFoundException("Cliente " + userName + " no encontrado."));
}
```

Hemos agregado a nuestro método, tres líneas nuevas en la parte superior del cuerpo. Simplemente estamos validando que el parámetro *"userName"* proveído por el cliente que consume el servicio esté conformado exactamente por tres caracteres. En el caso contrario, lanzaremos la nueva excepción creada con un mensaje de error.



Testeamos nuevamente el servicio desde POSTMAN enviando un "userName" con más de tres caracteres para provocar el error. Como pueden ver, el código de respuesta en el status es 400 Bad Request. Esto significa que cuando se lanzó la excepcion *"BadRequestExcepction"*, Spring la capturó y creó el *"ResponseEntity"* con el valor 400.

En esta respuesta no se está mostrando el error en sí al cliente. Es decir, no se muestra el mensaje de error con el cual se construye la excepcion. Tampoco se enviaba el error en la excepcion de *"ResourceNotFoundException"*.

Es importante y conveniente que el cliente que ejecuta las peticiones de nuestros servicios pueda tener un detalle más exacto de lo que está sucediendo. Por eso, Spring nos permite implementar una clase para interceptar el manejo de estas excepciones de manera más específica.

3.5.4 ControllerAdvice

Spring propone la creación de una clase Java con el propósito exclusivo de manejar el comportamiento de la aplicación para cada una de las excepciones que sean lanzadas durante el consumo de los servicios.

Esta clase debe tener la anotación @ControllerAdvice. Aquí se muestra cómo implementarla. La podremos crear dentro del paquete controllers.

```java
import org.springframework.web.servlet.mvc.method.annotation.ResponseEnt

import java.time.LocalDateTime;
import java.util.LinkedHashMap;
import java.util.Map;

@ControllerAdvice
public class RestResponseEntityExceptionHandler
  extends ResponseEntityExceptionHandler {

    @ExceptionHandler(BadRequestException.class)
    protected ResponseEntity<Object> handleBadRequest(
      RuntimeException ex, WebRequest request) {

        Map<String, Object> body = new LinkedHashMap<>();
        body.put("timestamp", LocalDateTime.now());
        body.put("message", ex.getMessage());
        body.put("Error",HttpStatus.BAD_REQUEST.toString());

        return new ResponseEntity<>(body, HttpStatus.BAD_REQUEST);
    }
}
```

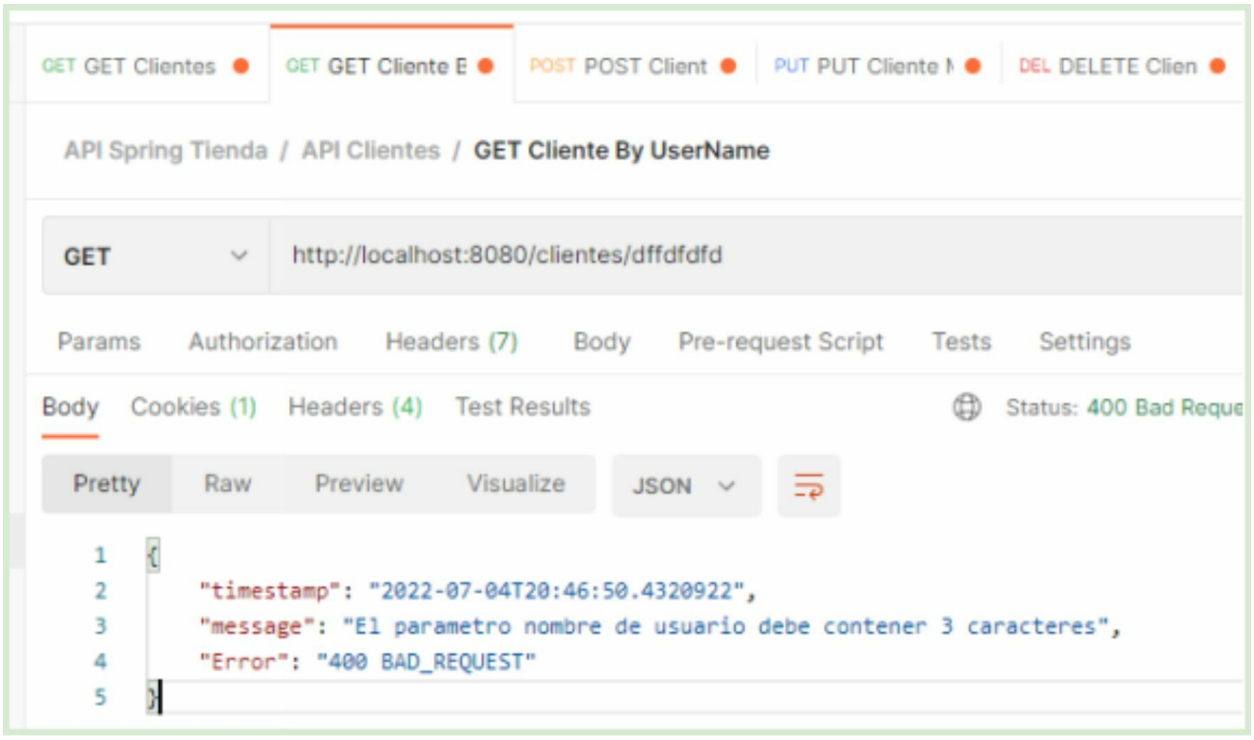
Como pueden verificar, la clase está decorada con la anotación @ControllerAdvice, lo que indica que esta clase estará destinada a manejar las respuestas de los servicios REST para las excepciones requeridas.

El método *"handleBadRequest"* manejara la respuesta de los servicios REST cada vez que se lance la excepcion *"BadRequestException"*. Esto se logra anotando el método con @ExceptionHandler(BadRequestException).

Ya dentro del método podemos hacer prácticamente lo que queramos siempre que retornemos un *"ResponseEntity"* válido. Para eso tenemos también la información necesaria desde los parámetros.

Para este caso, estamos devolviendo un *"ResponseEntity"* con código de status 400 y un pequeño body de respuesta que se traducirá en un JSON conteniendo algunos datos útiles. Entre ellos, el más destacado es el error de la excepcion que nosotros generamos a medida.

Ahora al ejecutar nuevamente desde POSTMAN podemos visualizar una respuesta mucho más descriptiva y detallada.



Observar que el error ahora es más contundente y específico. Indica realmente lo que está sucediendo de manera tal que el cliente puede tomar partido para corregir la petición de acuerdo a estos mensajes.

Cabe destacar, que se puede añadir mucha más información en esta respuesta, tanto como nosotros creamos útiles para aquellos clientes que usen nuestra API.

Internal Error Server 500

Este es el más temido de los errores, y a diferencia de los anteriores mencionados, se trata de un error como tal y no de una excepción. Este error, rara vez lo deberíamos gestionar nosotros desde el código ya que se produce automáticamente cuando tenemos un error de codificación, como puede ser un puntero nulo (*"NullPointerException"*) o cualquier tipo de anomalía como una Base de Datos caida, un Memory Leak, etc.

Otros códigos de respuesta

En esta sección hemos mencionado los típicos códigos de respuesta manejados en los servicios más frecuentes. No obstante, existe una gama más amplia de códigos de estado HTTP que las podemos consultar muy fácilmente en internet.

A medida que la complejidad de nuestra API crezca con el tiempo, es posible que debamos recurrir al manejo de excepciones más específicas y es altamente recomendable siempre buscar si el error que queremos manejar se encuentra contemplado en ese listado estándar de errores.

3.7 Capa controller - Otros tipos de media

En esta sección veremos otros tipos de devolución de contenido como alternativa al típico JSON.

3.7.1 Distintos tipos de respuesta

Si bien este libro está enfocado en la elaboración de una API REST donde el tipo “media” de dato más transportado es el estándar JSON, vamos a ver resumidamente otros tipos de media soportados por Spring que pueden ser útiles para algunos casos.

Para llevar a cabo estos ejemplos crearemos un controlador llamado *“ClienteRenderController”* en el cual haremos pruebas para retornar el cliente en distintos formatos.

3.7.2 Respuesta HTML

Crearemos el controlador *"ClienteRenderController"* dentro del paquete *"controllers"* con el siguiente método.

```java
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class ClienteRenderController {

    @GetMapping(value = "/clientes-html",produces = MediaType.TEXT_HTML_VALUE)
    public String getClienteAsHtml(){

        StringBuilder sb = new StringBuilder();
        sb.append("<html>");
        sb.append("<body>");
        sb.append(" <div><h1>Cliente</h1>");
        sb.append("  <ul>");
        sb.append("    <li>Nombre: Rafael Benedettelli</li>");
        sb.append("    <li>UserName: RBL</li>");
        sb.append("  </ul>");
        sb.append(" </div>");
        sb.append("</body>");
        sb.append("</html>");
        return sb.toString();
    }

}
```

Observemos que la directiva más contundente en este ejemplo es el atributo “produces” dentro de la anotación @GetMapping. Este atributo indica que la devolución de este servicio será entregada en formato HTML. Es importante que la implementación de esta devolución la implementemos en consonancia con el formato producido.

Como pueden ver, se crea un *“String”* que genera una serie mínima de etiquetas HTML para representar un cliente.

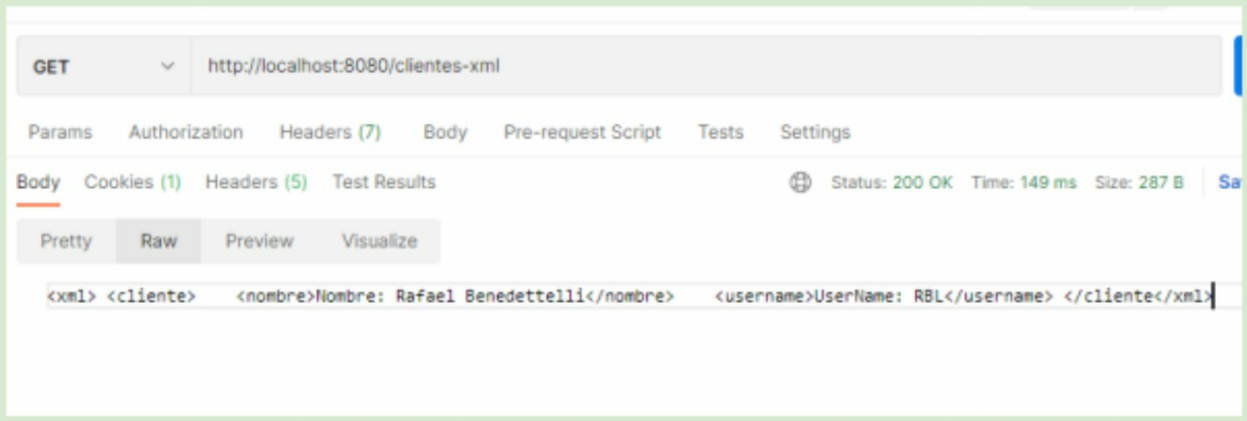
3.7.3 Respuesta XML

También, podremos entregar un cliente de acuerdo a este formato. Simplemente deberemos hacer los siguientes cambios en nuestro método.

```java
package edu.tienda.core.controllers;

import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class ClienteRenderController {

    @GetMapping(value = "/clientes-xml",produces = MediaType.APPLICATION_XML_VALUE)
    public String getClienteAsHtml(){

        StringBuilder sb = new StringBuilder();
        sb.append("<xml>");
        sb.append(" <cliente>");
        sb.append("    <nombre>Nombre: Rafael Benedettelli</nombre>");
        sb.append("    <username>UserName: RBL</username>");
        sb.append(" </cliente");
        sb.append("</xml>");
        return sb.toString();
    }

}
```

Observemos que realizamos las siguientes modificaciones. En primer lugar, la dirección del servicio ahora se llama *"/clientes-xml"*. Esta notación que emplea el guión para separar palabras en una URL es conocida como "Hyphen".

La modificación más importante por supuesto, es la que indica el "media type" de devolución, que para este caso es APPLICATION_XML_VALUE. Luego, se adapta el código para retornar el formato propuesto. Al probarlo desde POSTMAN podremos verificar que se resuelve la entrega en XML.



Existen muchos más tipos de media type que se pueden producir por los servicios de Spring. Incluso, tipos de extensión como PDF, imágenes PNG, Markdown, etc.

3.8 Capa controller - Configuration de base path

En esta sección veremos cómo configurar la URL base de nuestra API REST.

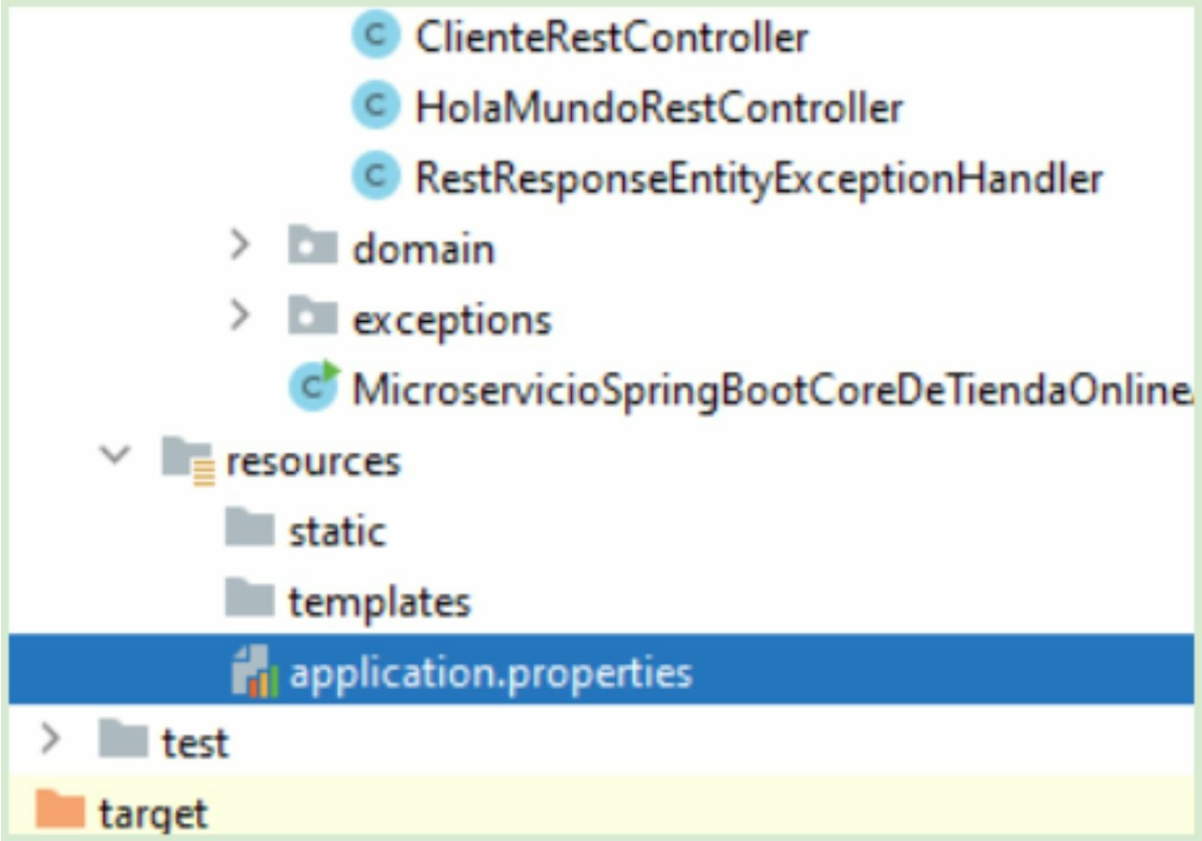
3.8.1 Application.properties

Este archivo de texto plano y de tipo clave-valor situado en la carpeta "*resources*" de nuestro proyecto es crucial para configurar ciertos ajustes de tipo global de nuestra solución. En este archivo, podremos configurar varios parámetros globales a medida, sin embargo, Spring ya propone algunas claves predefinidas para atacar ciertos propósitos.

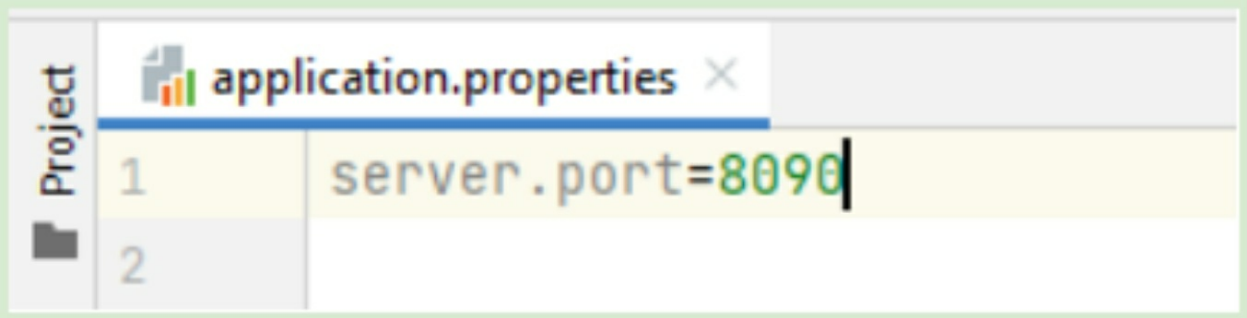
En principio, tengamos en cuenta que Spring Boot por default se ejecuta como servicio escuchando en el puerto 8080 (que es el predefinido por tomcat). Esta configuración es posible cambiarla simplemente agregando el siguiente "par valor" en el archivo de configuración *"application.properties"*.

3.8.2 Cambio de puerto

Localizamos el archivo *"application.properties"* que se encuentra situado en la carpeta *"resources"* y lo abrimos.



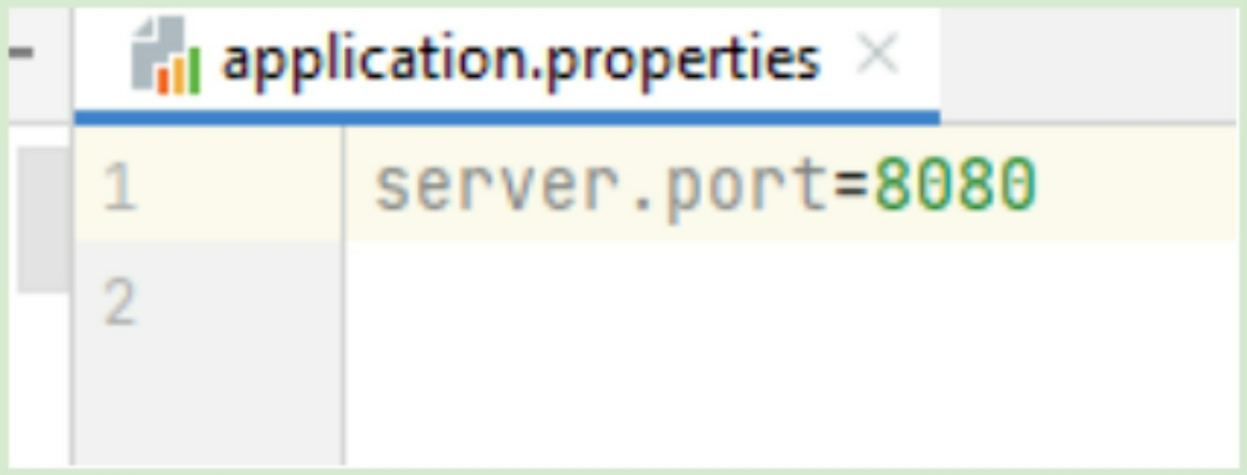
A continuación agregamos la clave *"server.port = 8090"* para cambiar el puerto de arranque.



Reiniciamos Spring y verificamos que tomcat ahora está escuchando en el puerto 8090.

```
] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicat:
] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initializatic
] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8090 (http) w:
] oSpringBootCoreDeTiendaOnlineApplication : Started MicroservicioSpringBootCoreDeTier
```

Ahora que ya entendimos cómo modificar el puerto de arranque, sugerimos cambiar este valor nuevamente a 8080 para que la configuración de nuestra colección de peticiones en POSTMAN no se vean alteradas.



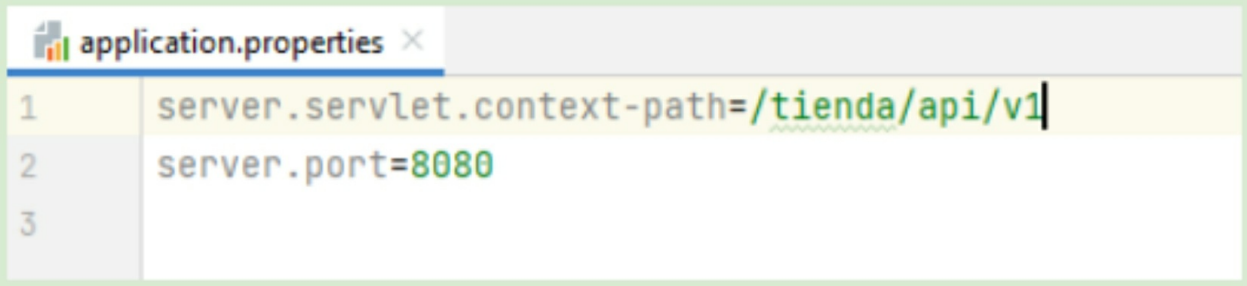
3.8.3 Cambio de base path

Actualmente nuestro base path es http://localhost:8080. Si quisiéramos consumir la API de clientes que hemos definido simplemente deberemos añadir a este base path el sufijo "/clientes" de manera tal de conformar la URL http://localhost:8080/clientes.

Si bien este es el base path que por defecto produce Spring, es recomendable modificarlo para generar una nomenclatura más acorde a los lineamientos definidos por las convenciones REST.

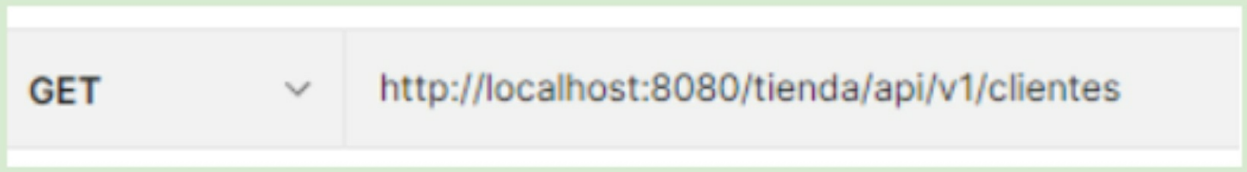
Una práctica sugerida es que la base de nuestra URL este nomenclada por el nombre general de la aplicación y seguido por la versión de la API expuesta hasta el momento. En función de esta regla, vamos a generar una URL base siguiendo este formato: http://localhost:8080/tienda/api/v1/. Como podemos observar, se puede identificar en la URL el nombre de la aplicación que es "tienda" seguido por "api" y la versión de la misma ("v1").

Para implementar esta nueva definición agregamos la siguiente clave a nuestro archivo de configuración *"application.properties"*.



La clave "server.serlet.context-path" indica que ahora la nueva URL de todo nuestro back-end es http://localhost:8080 seguida por el valor ***/tienda/api/v1***.

Al reiniciar Spring, deberemos atacar a nuestros servicios utilizando esta nueva dirección.



A partir de ahora, será necesario cambiar la dirección de las demás peticiones que habíamos definido en POSTMAN para que sigan funcionando correctamente.

Capítulo 4

4.1 Capa Services

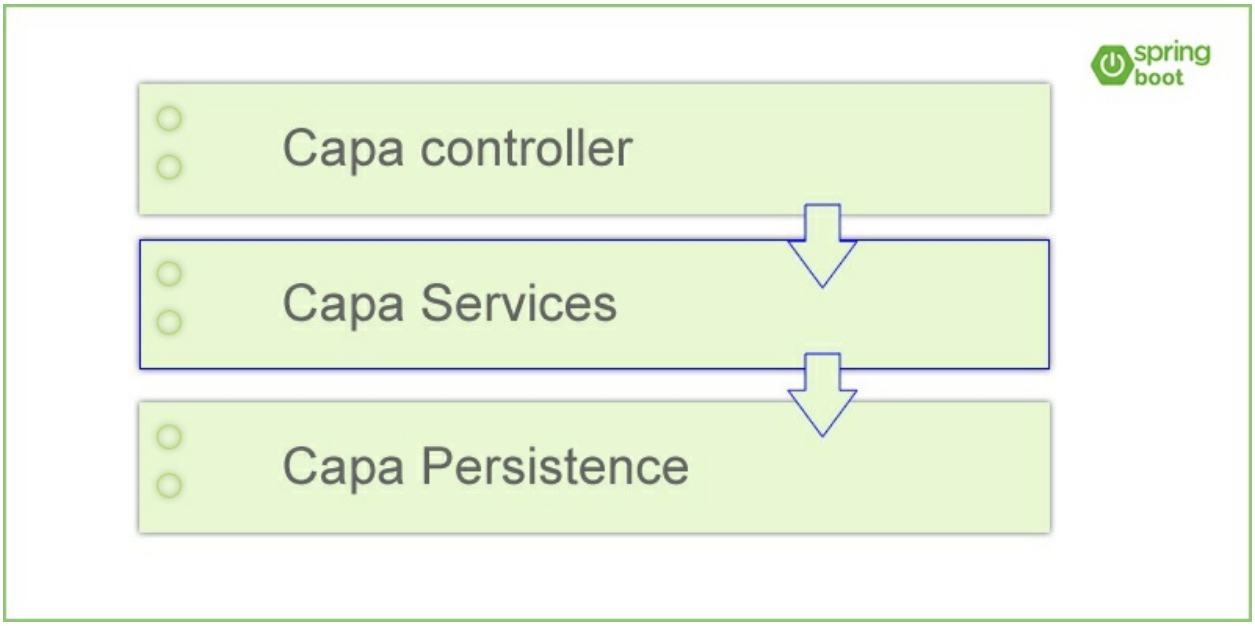
En este capítulo haremos una introducción a la capa de servicios y sus responsabilidades.

4.1.1 Introducción a la capa de servicios

Capa de servicios, Business Rule, Engine, Core, lógica de negocios son distintas formas de referirse a esta capa intermedia que es la más significativa de nuestra arquitectura, ya que básicamente, en ella, residirá nuestro programa como tal.

En esta capa se encuentra el grueso de nuestro software. Aquí se definen todos los algoritmos complejos para implementar la lógica de nuestro sistema. Por ejemplo, si se tratase de un sistema bancario, las transacciones entre cuentas, préstamos, etc, deberían implementarse en esta capa.

Mencionado esto, cabe destacar, que la capa de controlador debe estar enfocada sola y exclusivamente al manejo, exposición y presentación de datos de los servicios REST que la componen. Y bajo ninguna circunstancia debe estar poblada por código que resuelva algún problema intrínsecamente relacionado con el negocio. Simplemente debe obtener datos y presentarlos.



Entonces la capa de servicios (Service layer) es nuestro software en si, nuestro engine.

La idea de la construcción de un software en capas interconectadas ligeramente tiene como propósito la posibilidad de que estas sean “retráctiles” entre ellas. Es decir que cualquier cambio mayúsculo en una capa no se vea alterada por la capa superior o inferior. Esto se llama “bajo acoplamiento”. Un concepto muy importante, no solo de la ingeniería de software, sino también que se encuentra presente en la construcción de proyectos ingenieriles de orden civil, aeronáutico, militar e industrial.

Abordaremos este tema del bajo acoplamiento nuevamente en las próximas secciones.

4.2 Capa Services - Implementación

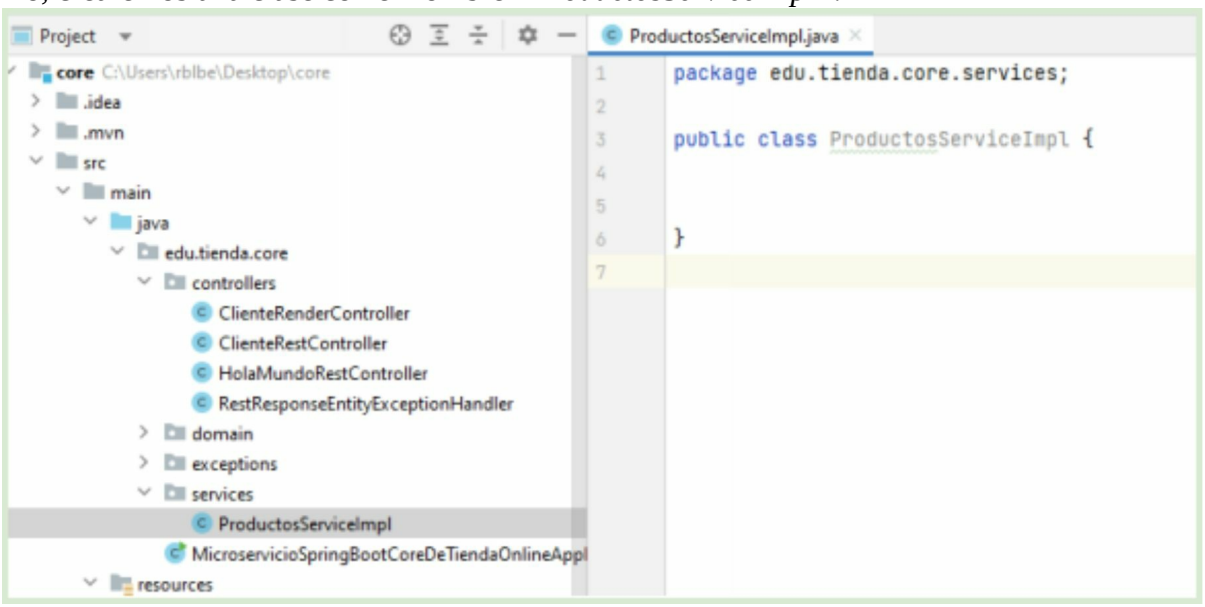
En esta sección programaremos nuestro primer servicio.

4.2.1 Implementación de un servicio

En esta sección implementaremos una simple clase Java de servicio y como siempre propone el libro, lo haremos de la forma más rústica posible. Como si no existiera Spring, para luego de a poco ir llevando esta implementación a la práctica más profesional recomendada.

Para nuestro primer acercamiento, implementar un servicio es tan simple como crear una clase Java con métodos que “hagan algo útil”.

Vamos a crear el paquete *“services”* dentro del paquete *“edu.tienda.core”*, y dentro del mismo, crearemos una clase con el nombre *“ProductosServiceImpl”*.



Esta clase tendrá como propósito la “gestión” de los productos de la aplicación.

En este punto necesitamos hacer una pausa y recordar que en la clase de controlador *“ClienteRestController”* hemos implementado un *“ArrayList”* con los clientes y dentro de cada método recorremos esta lista para filtrar, agregar, etc.

Esta implementación no es una buena práctica recomendada y cabe aclarar que lo hemos hecho solo con fines prácticos y pedagógicos, con la mera intención de focalizarnos en la capa de controlador y no abrir varias puertas al mismo tiempo.

Ahora vamos a realizar la implementación siguiendo las buenas prácticas. Primero implementaremos el servicio de productos y luego un controlador para que gestione los servicios REST de los mismos.

4.2.2 Implementando los métodos de ProductosServiceImpl

Vamos a implementar primero un almacén de productos similar a como lo habíamos hecho con clientes. Pero esta vez donde corresponde, en la clase de servicio de productos que hemos creado.

Primero crearemos una clase *"Producto"* en nuestro paquete *"edu.tienda.core.domain"* y le generamos un constructor con todos sus atributos y los respectivos "getters" y "setters".

```java
package edu.tienda.core.domain;

public class Producto {

    3 usages
    private Integer id;
    3 usages
    private String nombre;
    3 usages
    private Double precio;
    3 usages
    private Integer stock;


    3 usages
    public Producto(Integer id,String nombre, Double precio, Integer stock) {
        this.id = id;
        this.nombre = nombre;
        this.precio = precio;
        this.stock = stock;
    }

    public Integer getId() {
        return id;
    }
```

Ahora en la clase *"ProductosServiceImpl"* implementaremos un almacén con tres productos predefinidos. Definimos un *"ArrayList"* con tres productos creados directamente utilizando el constructor definido en la clase *"Producto"*.

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Optional;

public class ProductosServiceImpl {

        private List<Producto> productos = new ArrayList<>(Arrays.asList(
                new Producto( id: 1, nombre: "Smart TV", precio: 9000.0, stock: 3),
                new Producto( id: 2, nombre: "Pc Notebook", precio: 15000.0, stock: 10),
                new Producto( id: 3, nombre: "Tablet", precio: 8000.0, stock: 5)
        ));


}
```

Implementaremos los métodos “CRUD” para el alta, baja, modificación, eliminación y obtención de los productos. Primero implementaremos el método de recupero de todos los productos, que es el más simple.

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Optional;

public class ProductosServiceImpl {

    1 usage
    private List<Producto> productos = new ArrayList<>(Arrays.asList(
            new Producto( id: 1,  nombre: "Smart TV",  precio: 9000.0,  stock: 3),
            new Producto( id: 2,  nombre: "Pc Notebook",  precio: 15000.0,  stock: 10),
            new Producto( id: 3,  nombre: "Tablet",  precio: 8000.0,  stock: 5)
    ));


    public List<Producto> getProductos() {
        return productos;
    }


}
```

A este punto ya tenemos un servicio con alguna lógica mínima de excusa para utilizar. Vamos entonces, a crear el controlador que se va a usufructuar de la lógica de este servicio. Creamos en el paquete *"controllers"* una clase con el nombre *"ProductosControllerRest"* agregando como siempre las anotaciones @RestController y @RequestMapping("/productos")

Implementaremos un endpoint REST provisorio para el recupero de todos los productos del sistema.

```java
package edu.tienda.core.controllers;

import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/productos")
public class ProductoControllerRest {

    @GetMapping
    public ResponseEntity<?> getProductos(){

        return ResponseEntity.ok( body: null);
    }

}
```

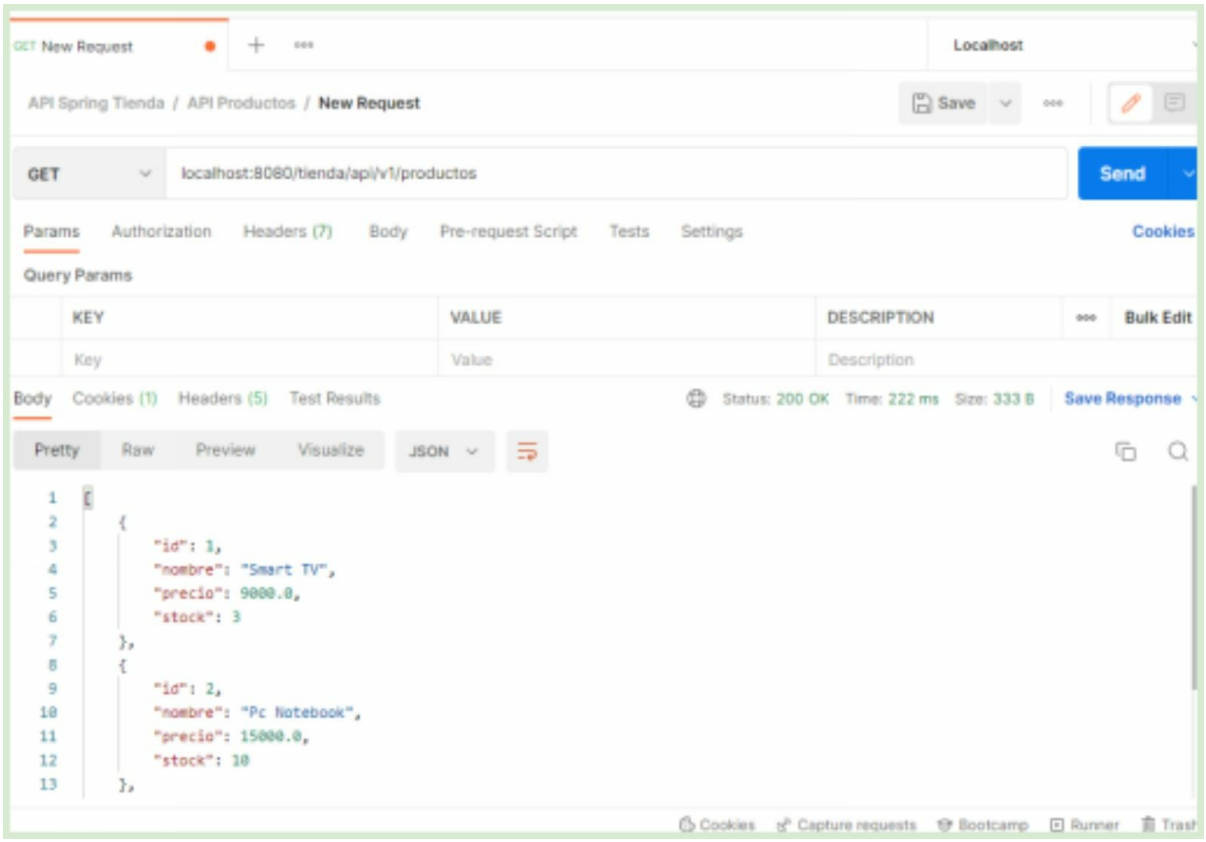
Tener en cuenta que implementamos un método que por el momento es provisorio ya que retorna *"null"*.

Aquí es donde vamos a unir la capa controladora con la capa de servicios, que es la verdadera responsable de gestionar los productos. En este caso, el controlador realiza una simple delegación al servicio para obtener los productos necesarios.

Para llevar esta conexión entre el controlador y el servicio, vamos a comenzar por lo más intuitivo al mejor estilo Java puro. Simplemente crearemos una referencia de *"ProductosServiceImpl"* en nuestra clase de controlador y la instanciamos. Luego, ejecutaremos el método *"getProductos"* para recuperar la lista y retornarla en el body del *"ResponseEntity"*.

```java
@RestController
@RequestMapping("/productos")
public class ProductoControllerRest {

    //Se instancia la clase de servicio al estilo "Java Puro"
    1 usage
    private ProductosServiceImpl productosService = new ProductosServiceImpl();

    @GetMapping
    public ResponseEntity<?> getProductos(){

        //Se recuperan todos los productos del servicio
        List<Producto> productos = productosService.getProductos();

        //Retornamos los productos del servicio en el body de la respuesta.
        return ResponseEntity.ok(productos);
    }

}
```

A continuación probaremos nuestro endpoint utilizando POSTMAN. Es recomendable crear una nueva carpeta en POSTMAN para la colección de nuestra API que tenga el nombre “API Productos”, similar a como lo hemos hecho con los clientes.



Lo importante aquí, más allá del funcionamiento, es el diseño. Hemos creado un controlador que delega en un servicio la obtención de todos los productos. Es sumamente importante notar la diferencia entre esta implementación y la de clientes, ya que el controlador anterior implementa toda la solución concentrada en una misma clase (“*ClientesRestController”*), mientras está, divide la codificación en dos clases que pertenecen a capas distintas, logrando de esta manera la división de responsabilidades apropiadas.

4.2.3 Programación orientada a interfaces

Vamos a realizar un ajuste a nuestra estructura de clases para cumplir con la práctica de implementar “orientado a interfaces”.

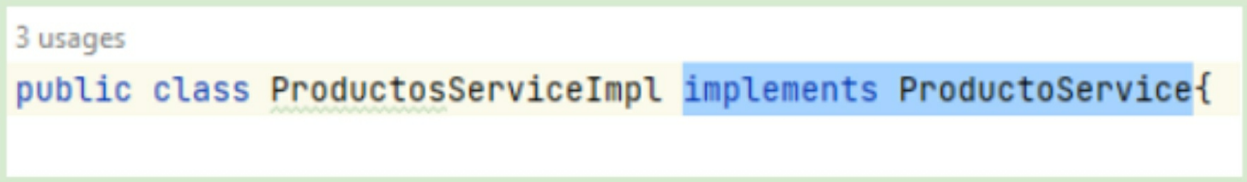
Las interfaces son clases totalmente abstractas y de índole declarativas en Java. Permite la definición de un contrato que deberá ser cumplido por aquellas clases concretas que la implementen.

Este tipo de definiciones nos ayudan a mantener un diseño mucho más flexible, ya que aprovecha el polimorfismo para la construcción y utilización de las clases. Este tipo de diseño, además, promueve el “bajo acoplamiento”, útil para la mantención y escalado del artefacto de software.

Crearemos entonces la interfaz *“ProductoService”* en el paquete *“edu.tienda.core.services”* y le definiremos el metodo *“getProductos”* de la siguiente manera.

```
package edu.tienda.core.services;

import edu.tienda.core.domain.Producto;

import java.util.List;

public interface ProductoService {

    public List<Producto> getProductos();
}
```

Haremos que la clase *“ProductosServiceImpl”* implemente esta interfaz que hemos creado.



Entonces la clase *"ProductosServiceImpl"* implementa la interface *"ProductService"*. Para aprovechar este nuevo diseño, realizaremos una pequeña modificación en la clase *"ProductosControllerRest"*.

Lo que hemos hecho es cambiar el tipo de variable de referencia de *"ProductosServiceImpl"* a *"ProductoService"*. Este tipo de construcción es polimórfica ya que define una variable con un tipo general y asigna como valor el constructor de una clase especializada.

Este manejo polimórfico nos brinda el beneficio de manejar diferentes implementaciones de servicios de productos unificadamente ya que estas siempre deberán cumplir con el contrato propuesto por la interface *"ProductosService"*.

Imaginémonos que en un futuro, los productos serán recuperados de otro servicio externo REST que se encuentre en la nube. En ese caso podríamos crear una clase concreta *"ProductosNubeService"* que implemente esta misma interfaz *"ProductosService"* con toda la lógica necesaria para obtener ese listado de productos desde la nube.

Este cambio o escalado no afectaría en gran medida el comportamiento de nuestra clase controladora *"ProductosControllerRest"*, ya que simplemente deberemos cambiar la construcción a *"private ProductService productService = new ProductosNubeService()"* y nuestro sistema funcionara igual pero con otra estrategia.

Este diseño orientado a interfaces que hemos implementado es una mejora, pero todavía la conexión entre el controlador y el servicio sigue siendo acoplada. Menos acoplada que antes, pero acoplada al fin. Este acoplamiento está dado porque en la clase controladora todavía tenemos una referencia explícita y sintáctica a la clase *"ProductosServiceImpl"* en la línea de instanciación.

4.2.4 Alto acoplamiento

Si bien hemos logrado separar las responsabilidades. Estas dos clases, *"ProductosControllerRest"* y *"ProductoServiceImpl"* están "fuertemente acopladas". Para decirlo de un modo más informal, están "enganchadas", y ese enganche, está dado debido a que la clase *"ProductosControllerRest"* está a cargo de la creación del objeto de servicio.

Para lograr que la capa controladora y la capa de servicio mantengan un "fino enlace" de conexión, la clase *"ProductosControllerRest"* no debería tener conocimiento alguno de la existencia de la clase *"ProductosServiceImpl"*.

Si somos lo suficientemente buenos programando en Java podríamos animarnos a crear una clase "intermedia" que juegue un rol de "proxy". Esta clase de tipo "proxy" sería la encargada de la factoría de la clase *"ProductosServiceImpl"* y resolveriamos el problema.

Afortunadamente Spring se encargará de esa factoría por nosotros. Para eso hará uso de unas simples anotaciones.

4.2.5 Inyección de dependencias con Spring

Con el objetivo principal de separar y desacoplar ya totalmente la clase de servicio con la de controlador, vamos hacer uso de las inyecciones de dependencias de Spring. Simplemente decoramos nuestra clase *"ProductosServiceImpl"* con la anotación @Service como se muestra a continuación.

```
2 usages
@Service
public class ProductosServiceImpl implements ProductoService
```

Esta anotación hará que la clase quede registrada por Spring en su contenedor de dependencias. De esta manera será Spring el encargado de producir instancias de esta clase automáticamente. Este tipo patrón se llama "inversión del control" y también se lo conoce como patrón "Hollywood"

Este simpático nombre, es usado ya que hace mención a una frase muy utilizada en el mundo de los castings artísticos "No nos llames, nosotros te llamaremos".

En el mundo del desarrollo, esta frase se traduce como "No instancies objetos, nosotros lo

haremos por ti”. Y es justamente Spring es quien se encargará de hacerlo por nosotros.

Ahora que ya tenemos registrada la clase en el contenedor de dependencias de Spring, vamos a modificar la clase de controlador para inyectar automáticamente esa instancia con el uso de la anotación @Autowired.

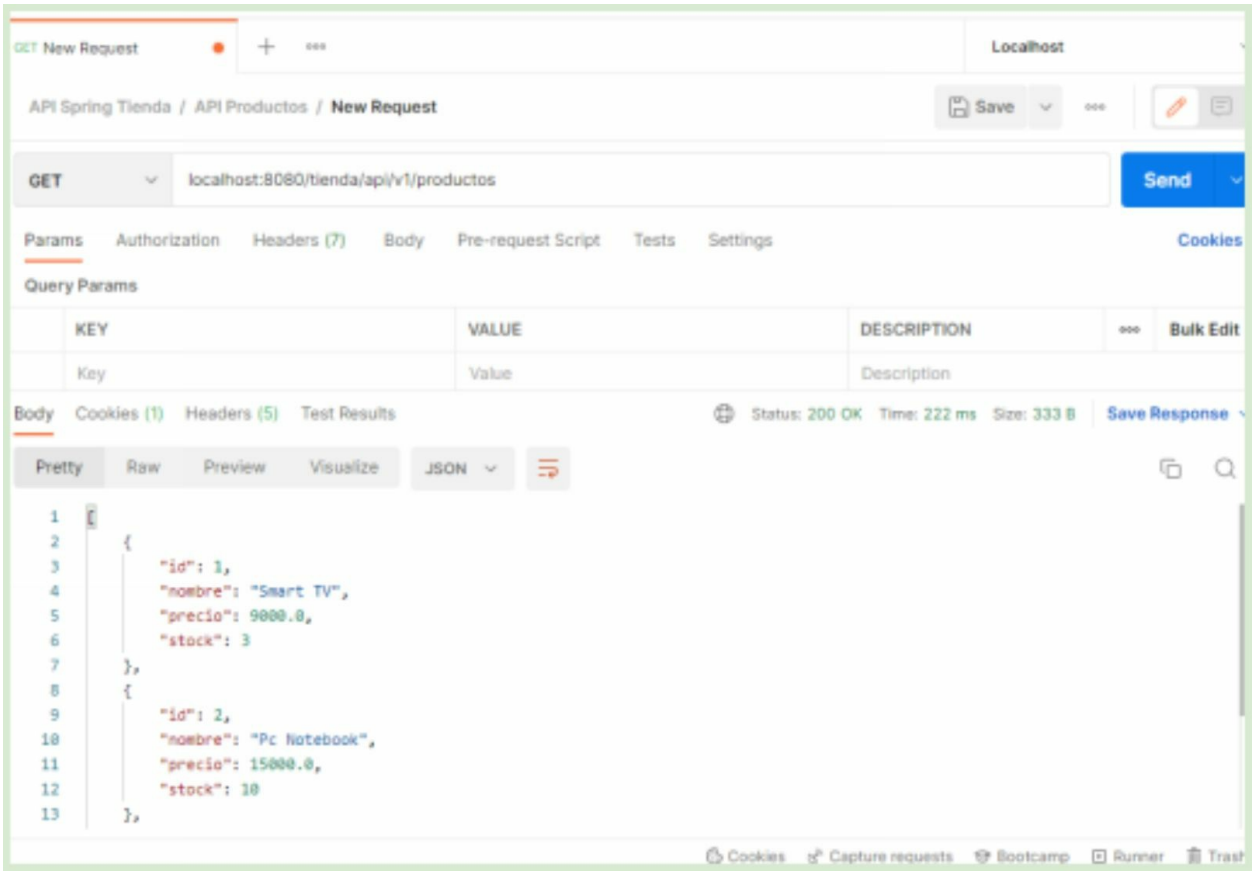
```java
@RestController
@RequestMapping("/productos")
public class ProductoControllerRest {

    //Se instancia la clase de servicio al estilo "Java Puro"
    1 usage
    @Autowired
    private ProductoService productosService;

    @GetMapping
    public ResponseEntity<?> getProductos(){

        //Se recuperan todos los productos del servicio
        List<Producto> productos = productosService.getProductos();

        //Retornamos los productos del servicio en el body de la respuesta.
        return ResponseEntity.ok(productos);
    }

}
```

Observar que hemos realizado dos pequeñas modificaciones:

1. La primera es eliminar la creación de la instancia de *"ProductosServiceImpl"*. Es justamente porque queremos que ahora Spring se encargue de esta factoría.
2. En segundo lugar, y justamente para que Spring sepa que tiene que crear una instancia de *"ProductosServiceImpl"* y guardarla en la referencia *"ProductoService"*, decoramos al atributo con la anotación @Autowired.

La anotación @Autowired indica a Spring que tiene que buscar en su contenedor de inyecciones una clase que implemente a la interfaz que decora. Luego Spring, creará una instancia de *"ProductosServiceImpl"* y la guardará en la referencia *"ProductoService"* de manera transparente tanto al programador como a la clase controladora.

El simple cambio de que Spring sea el encargado de instanciar los objetos por nosotros, plasma un diseño con muy bajo acoplamiento entre dos clases. Probemos nuestro servicio nuevamente con POSTMAN para comprobar que su funcionamiento no se ha alterado a pesar del nuevo diseño que hemos implementado.


GET New Request
Localhost
API Spring Tienda / API Productos / New Request
Save
GET
localhost:8080/tienda/api/v1/productos
Send
Params
Authorization
Headers (7)
Body
Pre-request Script
Tests
Settings
Cookies
Query Params
KEY
VALUE
DESCRIPTION
Bulk Edit
Key
Value
Description
Body
Cookies (1)
Headers (5)
Test Results
Status: 200 OK
Time: 222 ms
Size: 333 B
Save Response
Pretty
Raw
Preview
Visualize
JSON
[
{
"id": 1,
"nombre": "Smart TV",
"precio": 9000.0,
"stock": 3
},
{
"id": 2,
"nombre": "Pc Notebook",
"precio": 15000.0,
"stock": 10
},
Cookies
Capture requests
Bootcamp
Runner

4.2.6 Servicio de productos alternativo

Supongamos ahora, que ha pasado un tiempo razonable desde que este software se encuentra en producción. Y con el tiempo ha ocurrido un pequeño cambio en nuestro negocio.

Ahora se propone una nueva “feature” en nuestro sistema para que esa lista de productos se puedan obtener desde un archivo JSON local y no desde memoria.

Para implementar dicha “feature”, hay dos caminos posibles. La primera, podría ser la de modificar el código fuente de la clase *“ProductosServiceImpl”* para que en lugar de buscar los productos en memoria lo haga desde el nuevo archivo JSON.

Sin embargo, esta estrategia se podría ver afectada por los siguientes escenarios. En primer lugar, en un proyecto real, la clase *“ProductosServiceImpl”* podría contener una cantidad extensa de líneas de código ya que pudo haber crecido en lógica y complejidad con el paso del tiempo.

Y en segundo término, si modificamos directamente esta clase ya no dispondremos de su versión original en el caso de que se decida un “rollback” para retornar al sistema a su implementación original.

Por lo tanto, una buena decisión es la creación de un Servicio alternativo que implemente la nueva lógica para la obtención de productos.

4.2.7 Implementando el servicio alternativo

Primero crearemos un archivo con nombre *"productos.json"* en la carpeta *"resources"* de nuestro proyecto con el siguiente contenido.

```
[
  {
    "id": 1,
    "nombre": "Play Station",
    "precio": 20000.0,
    "stock": 3
  },
  {
    "id": 2,
    "nombre": "Mouse",
    "precio": 1000,
    "stock": 10
  },
  {
    "id": 3,
    "nombre": "Monitor",
    "precio": 10000.0,
    "stock": 5
  }
]
```

Creamos la clase *"ProductosServiceJSONImpl"* en el paquete *"edu.tienda.core.services"* y haremos que también implemente la interfaz *"ProductoService"*. La nueva clase tendrá el siguiente aspecto:

Observar que el método *"getProductos"* tiene una implementación provisoria. Vamos a implementar la lógica de negocios de ese método que básicamente tendrá que hacer un "lookup" del archivo *"productos.json"* y cargarlos en una lista.

Pero primero crearemos un constructor "default" vacío en la clase *"Producto"*. Esto es necesario para que la librería Jackson pueda construir instancias de esta clase cuando ejecute la deserialización del archivo JSON.

```java
public class Producto {

    3 usages
    private Integer id;
    3 usages
    private String nombre;
    3 usages
    private Double precio;
    3 usages
    private Integer stock;


    public Producto(){

    }
```

Pasamos a implementar el cuerpo del método "*getProductos*" de la clase

“*ProductosServiceJSONImpl”* de esta manera.

```java
package edu.tienda.core.services;

import ...

@Service
public class ProductosServiceJSONImpl implements ProductoService{

    1 usage
    public List<Producto> getProductos() {
        List<Producto> productos;
        try {

            productos = new ObjectMapper()
                    .readValue(this.getClass().getResourceAsStream( name: "/productos.json"),
                            new TypeReference<List<Producto>>() {});

            return productos;

        }catch (IOException e) {
            throw new RuntimeException(e);
        }

    }


}
```

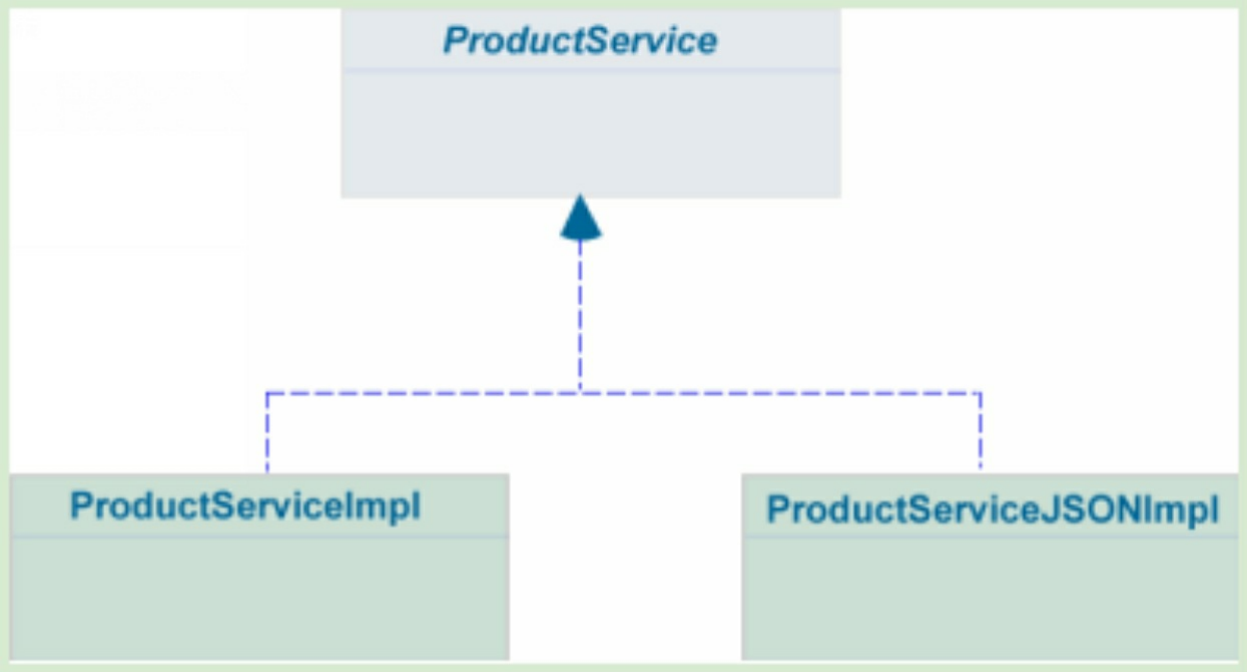
Este código situado en la imagen superior es el que carga el archivo JSON de productos desde el classpath del proyecto y lo deserializa a una lista de productos.

De todo este trabajo sucio se encarga la librería Jackson que es el conversor JSON-JAVA y viceversa que utiliza por definición Spring. En muchos casos lo hace de forma interna y transparente, como por ejemplo cuando resuelve las peticiones REST.

Observar que este nuevo "java bean" es un servicio alternativo que también implementa la interfaz *"ProductoService"* y también está decorada con la anotación @Service para que Spring la registre en su contenedor de dependencias.

Ahora cuando tratemos de reiniciar Spring Boot obtendremos una excepción en tiempo de arranque. Este error se producirá porque tenemos registradas dos implementaciones de la interfaz *"ProductoService"* y el inyector de independencias es incapaz de resolver cuál de ellas dos es la que se deberá inyectar en la variable definida en el controller con @Autowired.

En este momento disponemos de la siguiente estructura jerárquica donde *"ProductoServiceImpl"* y *"ProductoServiceJSONImpl"* implementan *"ProductoService"*.



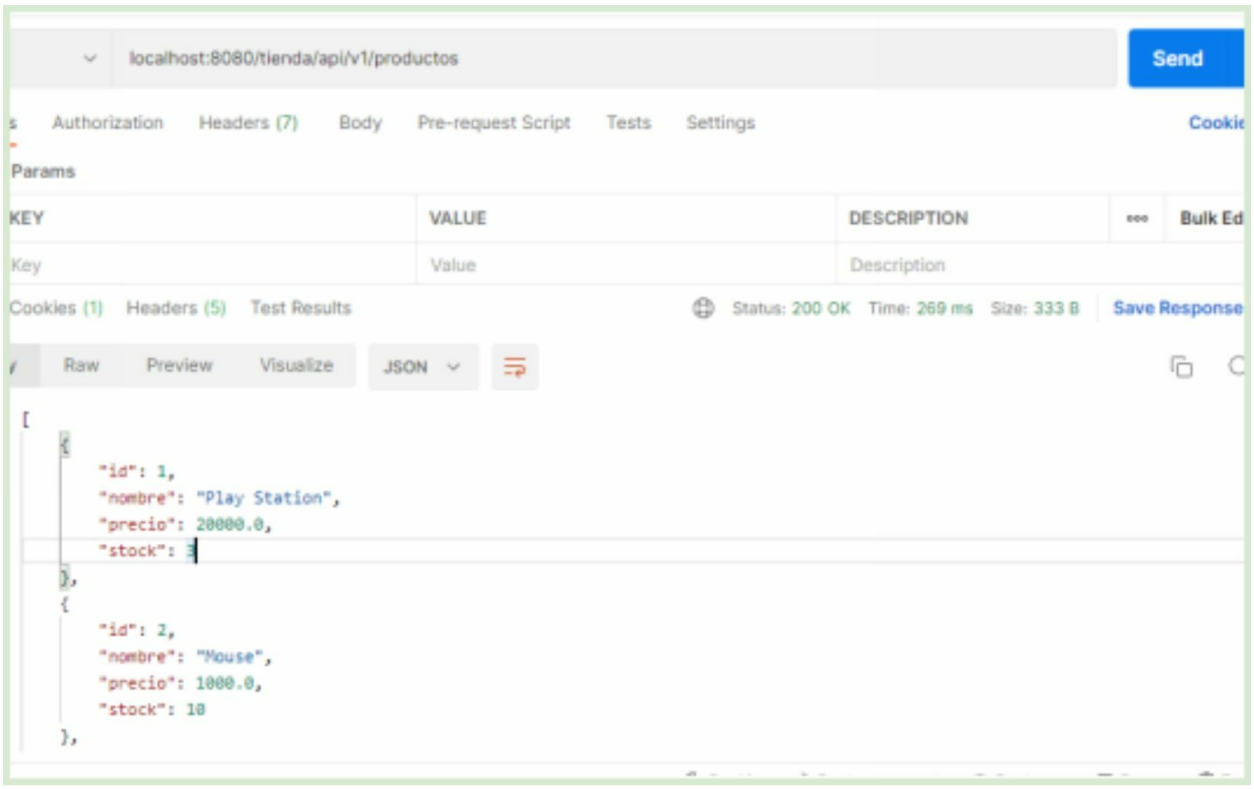
Como mencionamos anteriormente, Spring no puede resolver que implementación de las dos inyectar en la variable *"ProductoService productoService"* de la clase *"ProductoController"*.

Para darle un pequeño guiño y ayuda a Spring, vamos a diferenciar con una muy sutil anotación la clase *"ProductoServiceJSONImpl"* que es en donde hemos implementado la nueva feature, y que por lo tanto, es la que queremos que inyecte Spring. Decoramos entonces, la clase *"ProductoServiceJSON"* con la anotación @Primary.

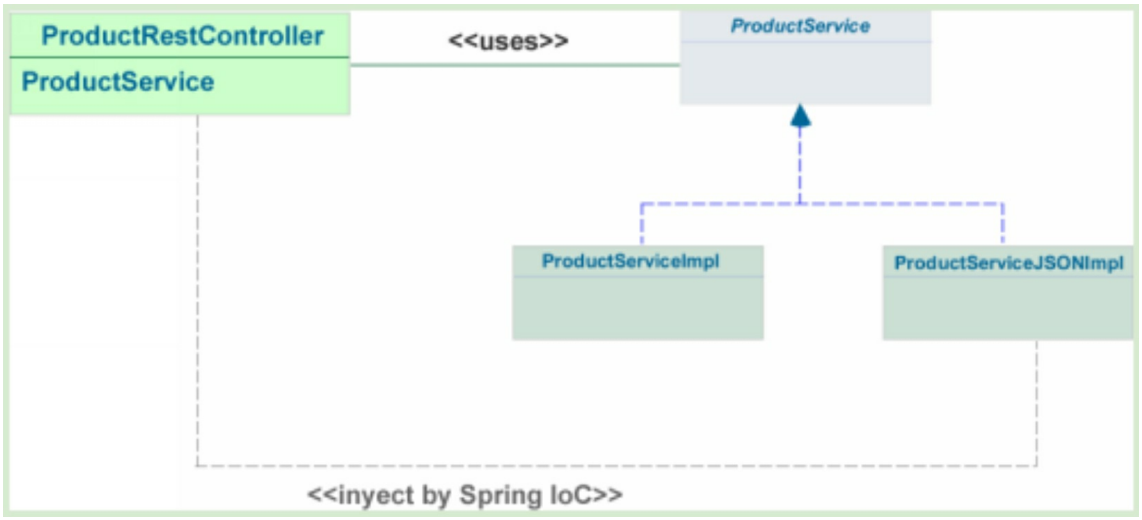
```
@Primary
@Service
public class ProductosServiceJSONImpl implements ProductoService
```

@Primary cómo lo dice la palabra distingue a un bean de otro indicando que el que está marcado con esta anotación debe tener prioridad para la inyección por sobre otros servicios "hermanos". Es decir, servicios que implementan una misma interfaz. Ahora al iniciar Spring podremos verificar que ya no se producirá la excepción anterior y arrancará sin problemas.

Ejecutamos desde POSTMAN el mismo servicio para comprobar que ahora la nueva lista de productos es la que hemos definido en el archivo JSON local.



En el siguiente diagrama de clases representamos visualmente el nuevo diseño de implementación.



De acuerdo a este diagrama, la clase *"ProductoRestController"* sólo tiene conocimiento de la interfaz *"ProductoService"* pero es, por detrás de escena Spring, quien se encarga de proveer una instancia de la clase *"ProductoServiceJSONImpl"*.

Es importante aclarar que la anotación @Service registra clases en el contenedor de dependencias Spring que luego generará un solo objeto de la misma durante toda la sesión. Esto significa que Spring inyecta estos beans de acuerdo al patrón "Singleton" (un objeto

solo por clase).

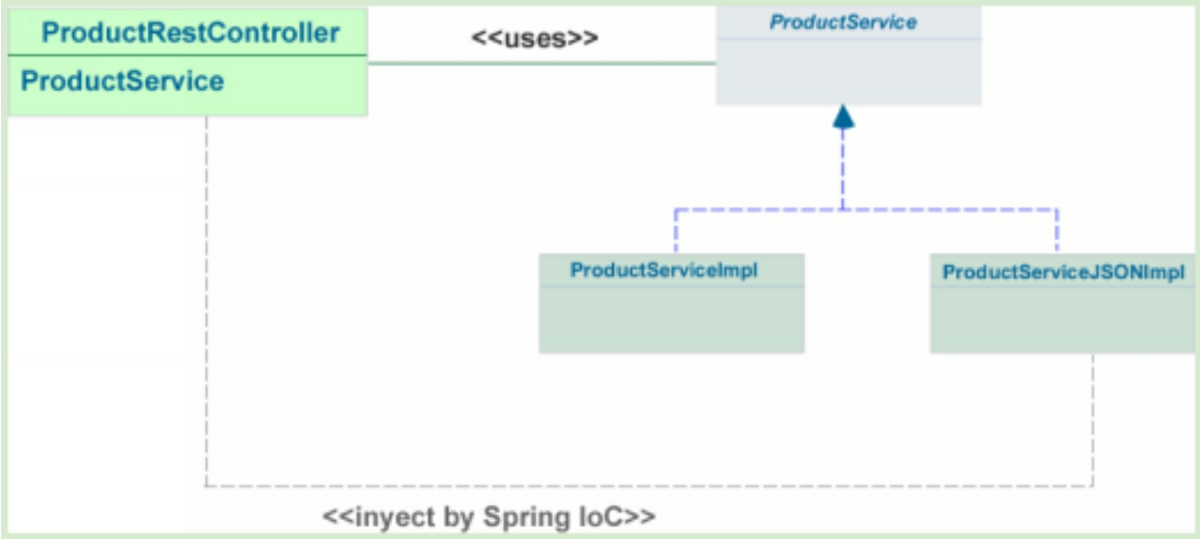
Así como hemos implementado en *"ProductoServiceImpl"* el método para la devolución de todos los productos, también es en esta clase de servicio donde se deberán implementar los demás métodos para el alta, baja y modificación.

4.3 Capa Services - Tipos de Autowired

En este capítulo implementaremos diferentes tipos de autowired.

4.3.1 Autowired por calificación

De acuerdo a nuestro esquema donde tenemos dos servicios que implementan la interfaz *"ProductoService"*.



Hemos aplicado un diferenciador con la anotación @Primary sobre la clase *"ProductoServiceJSONImpl"* para ayudar a Spring a distinguir cual de los dos servicios debería inyectar en la clase *"ProductoRestController"*.

@Primary es un tipo de diferenciador entre otros tantos, que en este caso simplemente indica que un "bean" tiene prioridad por sobre otro. Se deduce, que por supuesto, si el bean *"ProductoServiceImpl"* también tuviese la anotación @Primary volveríamos a incurrir en el mismo problema de ambigüedad. Ya que Spring no podría diferenciar cual de los dos servicios tiene prioridad.

Otro tipo de mecanismo para diferenciar dos o más beans que implementan la misma interfaz es mediante la anotación @Qualifier. Esta anotación permitirá a Spring resolver la inyección de acuerdo al nombre del bean. Primero y ante todo, debemos asignarle un nombre, un alias, a nuestros dos servicios. El primero lo denominaremos con el alias "JSON" de la siguiente manera.

```
@Service("JSON")
public class ProductosServiceJSONImpl implements ProductoService
```

Mientras que al segundo bean le asignaremos el alias "MEMORY".

```java
@Service("MEMORY")
public class ProductosServiceImpl implements ProductoService{
```

Como pueden observar ahora nuestros dos servicios tienen un nombre asignado. Gracias a esta nominación, desde el controlador podremos indicar de forma explícita cuál de los dos beans se debe inyectar. Lo haremos de la siguiente manera.

```java
@RestController
@RequestMapping("/productos")
public class ProductoControllerRest {

    //Se instancia la clase de servicio al estilo "Java Puro"
    1 usage
    @Autowired
    @Qualifier("MEMORY")
    private ProductoService productosService;
```

Hemos agregado una nueva anotación llamada @Qualifier decorando el atributo *"productosService"* que indica por nombre, cuál de los beans disponibles en el contenedor deberá inyectarse dentro de esta variable. Es entonces que Spring inyectará el bean *"ProductosServiceImpl"*. Observar que @Qualifier tiene prioridad por sobre @Primary.

4.3.2 Autowired condicional

En algunos casos, puede ser útil inyectar un bean u otro de acuerdo a alguna condición preestablecida en la configuración de nuestro sistema. Por ejemplo, podríamos generar una clave en nuestro archivo de configuración *"application.properties"* de la siguiente manera.

```
server.servlet.context-path=/tienda/api/v1
server.port=8080

productos.estrategia=EN_MEMORIA
```

Hemos definido una clave con el nombre *"productos.estrategia"* y con el valor *"EN_MEMORIA"*. Básicamente, estamos indicando con cierto dialecto arbitrario que la estrategia para recuperar nuestros productos será desde memoria. Es decir que quisiéramos recuperarlos con el servicio *"ProductoServiceImpl"* que obtiene los productos desde una lista Java. Pero, con esto, por supuesto, no es suficiente ya que Spring desconoce totalmente este nuevo par-valor configurado.

Para eso deberemos agregar la anotación "@ConditionalOnProperty" sobre los dos beans de servicios.

```
@Service("MEMORY")
@ConditionalOnProperty(
        value="productos.estrategia",
        havingValue = "EN_MEMORIA")
public class ProductosServiceImpl implements ProductoService{
```

Hemos agregado esta anotación con dos atributos. *"Value"* y *"HavingValue"*. Llevado a un idioma más coloquial, estamos instruyendo a Spring para que inyecte este bean siempre que la propiedad *"productos.estrategia"* definida en el *"application.properties"* tenga el valor *"EN_MEMORIA"*.

Haremos algo similar en el bean *"ProductosServiceJSONImpl"*.

```java
@Service("JSON")
@ConditionalOnProperty(
        value="productos.estrategia",
        havingValue = "EN_JSON")
public class ProductosServiceJSONImpl implements ProductoService{
```

En este caso hemos agregado la misma anotación donde se indica que para este caso se inyectara este bean siempre que la propiedad *"productos.estrategia"* tenga el valor *"EN_JSON"*.

Por último, deberemos remover la anotación @Qualifier en nuestro controller para que no superponga prioritariamente a la anotación @ConditionalOnProperty.

```java
@RestController
@RequestMapping("/productos")
public class ProductoControllerRest {

    //Se instancia la clase de servicio al estilo "Java Puro"
    1 usage
    @Autowired
    private ProductoService productosService;
```

Si reiniciamos Spring y ejecutamos la prueba en POSTMAN, verificamos que obtendremos la lista de productos en memoria dado que el valor de la clave "productos.estrategia" es igual a *"EN_MEMORIA"*. Luego podemos probar cambiar ese valor a *"EN_JSON"* y reiniciar Spring para verificar que la nueva devolución son aquellos productos definidos en el archivo JSON de nuestro proyecto.

Este mecanismo es bastante eficaz para aquellos escenarios donde nuestro sistema implementa distintas estrategias para una misma API. En este escenario, la decisión de inyección está comandada por el valor de una clave que reside en nuestro archivo de configuración principal de Spring Boot.

Spring está dotado de una amplia gama de condiciones para la inyección de beans que son un tanto más avanzados y minuciosos. Spring permite la inyección de beans en función de variables de entorno del sistema, versión de JRE que corre en el equipo, etc.

4.4 Capa Services - Inicialización Lazy

En esta sección veremos cómo modificar el mecanismo de inicialización de nuestros beans.

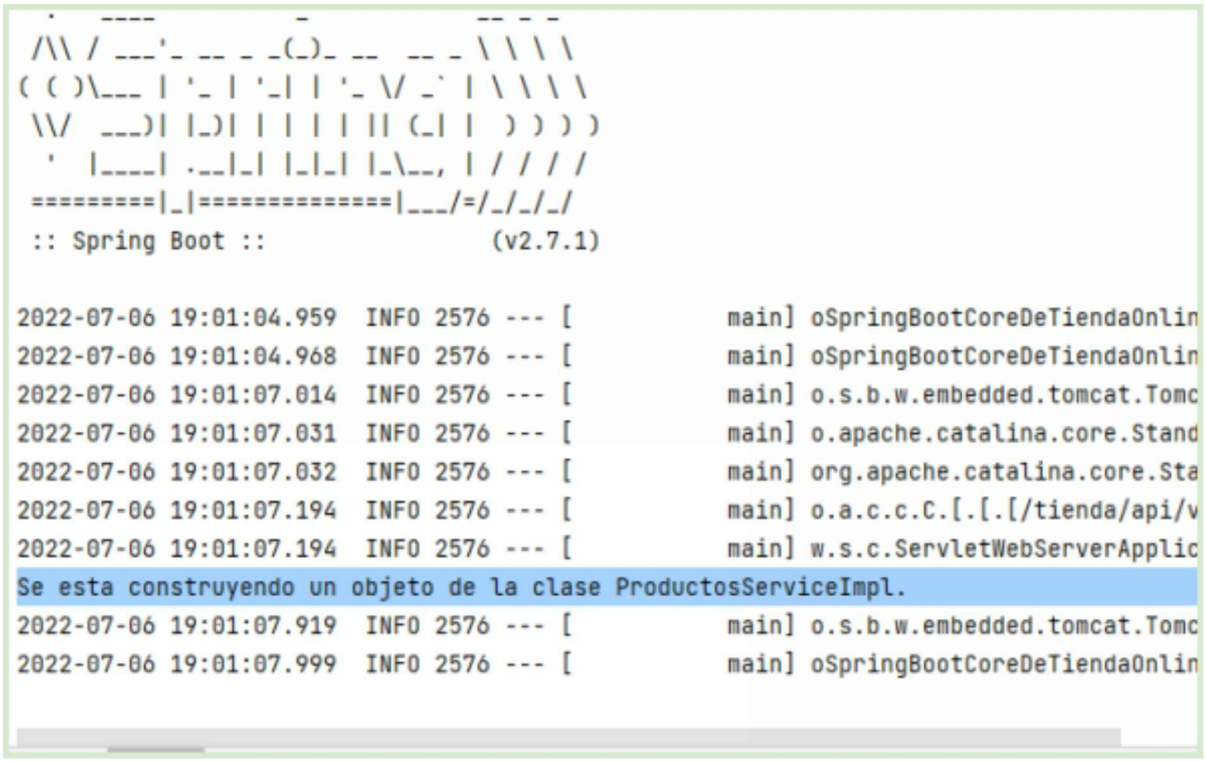
4.4.1 Mecanismo de inicialización por defecto

Spring, por defecto, crea los objetos de nuestros beans registrados en el contenedor de dependencias de forma “prematura”. Esto significa, que Spring iniciará los beans en el momento de arranque de la aplicación, osea, cuando hacemos el bootstrap.

Generalmente, esta será la opción de inicio para nuestros beans que vamos a pretender para la mayoría de servicios y componentes. No obstante pueden existir casos peculiares donde no queramos cargar en memoria ciertos beans que a priori sabemos que no van a ser muy demandados por el usuario o cliente de la API.

Para comprobar el inicio de beans prematuros, vamos a crear un constructor en nuestro servicio *“ProductoServicioImpl”* con una pequeña traza para indicar que se está creando.

En este caso hemos agregado un constructor simple sin argumentos con una pequeña traza. Recordemos que nosotros no somos los que creamos una instancia de esta clase, sino que lo hace Spring. Ahora reiniciamos Spring y verificamos, que se imprimirá esa traza durante el arranque de la aplicación.



Como podemos verificar, Spring creó una instancia de la clase *"ProductoServiceImpl"* y la inyectó en el controlador. Y esta acción la llevó a cabo durante el arranque, es decir de forma prematura.

Si el endpoint API REST para recuperar nuestros productos, que es el que utiliza este servicio, no tuviese un uso frecuente por los usuarios de nuestra aplicación final, estaremos cargando en memoria un bean que ocupa un espacio de manera ineficiente. Para este tipo de escenario podremos anotar la inyección para que utilice el mecanismo con retardo.

4.4.2 Mecanismo de inicialización con retardo

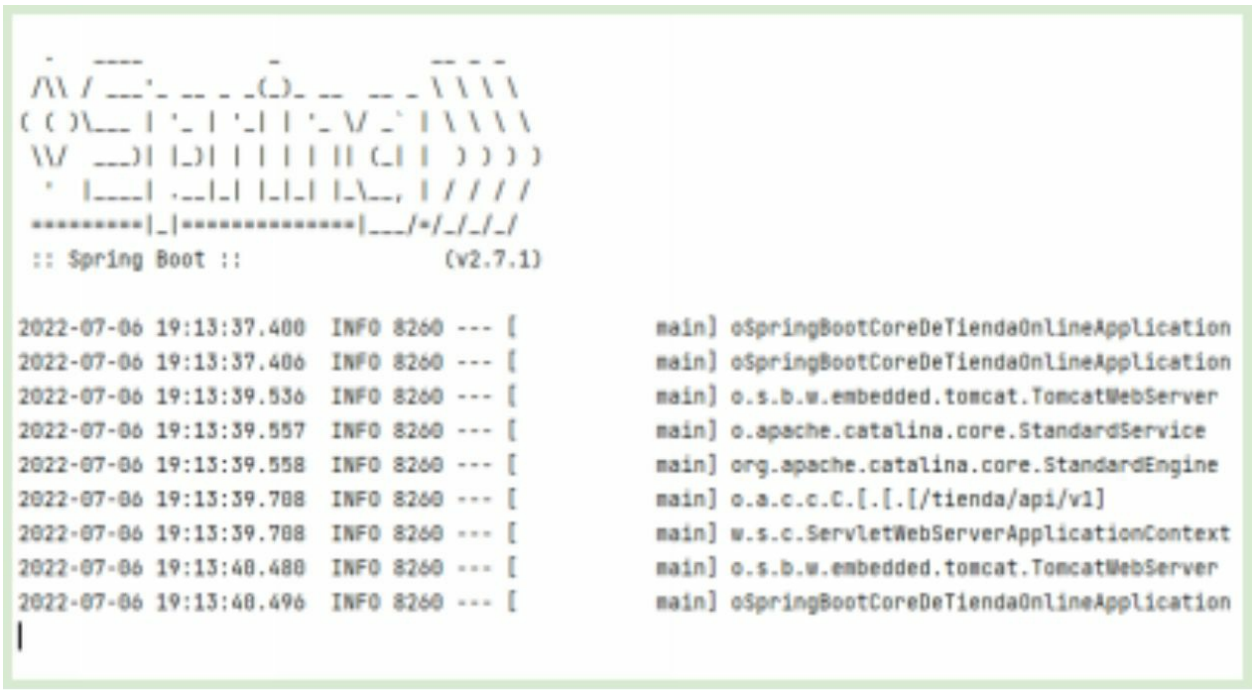
Este mecanismo, en oposición con el prematuro, inicializa el bean (crea una instancia de la clase *"ProductoServiceImpl"*) sólo bajo demanda. Es decir, solo cuando se ejecute el endpoint REST que hace uso del servicio. Para probar este mecanismo anotamos la referencia en el controller con la anotación @Lazy.

```
@RestController
@RequestMapping("/productos")
public class ProductoControllerRest {

    //Se instancia la clase de servicio al estilo "Java Puro"
    1 usage
    @Autowired
    @Lazy
    private ProductoService productosService;
```

También es necesario anotar @Lazy sobre el bean que queremos que inicialice con retardo.

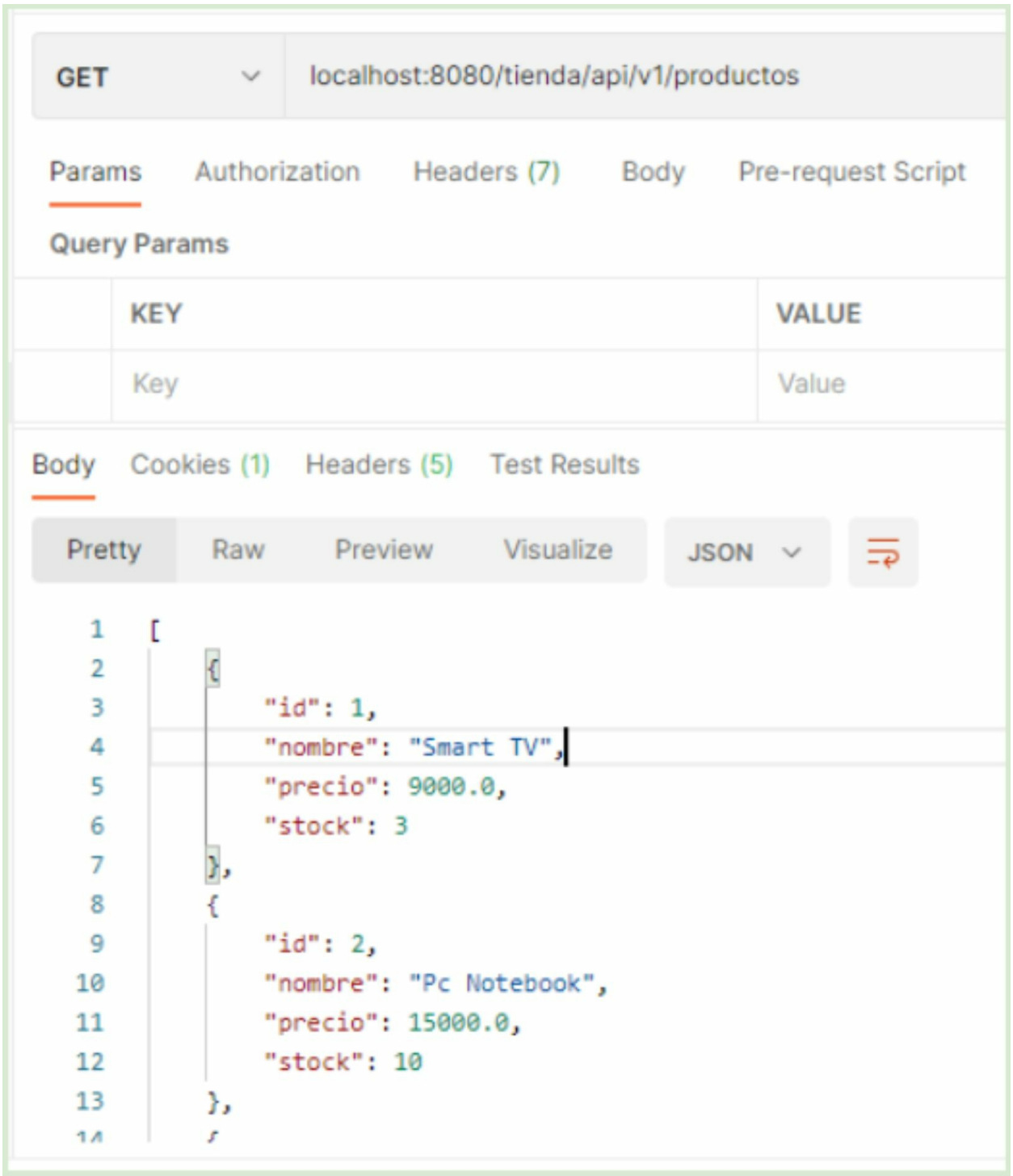
```
@Lazy
@Service("MEMORY")
@ConditionalOnProperty(
        value="productos.estrategia",
        havingValue = "EN_MEMORIA")
public class ProductosServiceImpl implements ProductoService{
```

Ahora al reiniciar Spring, observaremos que ya no se vuelve a imprimir la traza "Se está construyendo un objeto de la clase *ProductoServiceImpl*". Esto es debido a que este bean ya no se inicializa en tiempo de arranque, sino que se va a inicializar cuando ejecutemos el endpoint para obtener los productos. Ósea, bajo demanda.



```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.7.1)

2022-07-06 19:13:37.400  INFO 8260 --- [           main] oSpringBootCoreDeTiendaOnlineApplication
2022-07-06 19:13:37.406  INFO 8260 --- [           main] oSpringBootCoreDeTiendaOnlineApplication
2022-07-06 19:13:39.536  INFO 8260 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer
2022-07-06 19:13:39.557  INFO 8260 --- [           main] o.apache.catalina.core.StandardService
2022-07-06 19:13:39.558  INFO 8260 --- [           main] org.apache.catalina.core.StandardEngine
2022-07-06 19:13:39.708  INFO 8260 --- [           main] o.a.c.c.C.[.[.[/tienda/api/v1]
2022-07-06 19:13:39.708  INFO 8260 --- [           main] w.s.c.ServletWebServerApplicationContext
2022-07-06 19:13:40.480  INFO 8260 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer
2022-07-06 19:13:40.496  INFO 8260 --- [           main] oSpringBootCoreDeTiendaOnlineApplication
```

Ejecutamos desde POSTMAN el endpoint para recuperar todos los productos.



Lo más importante aquí es observar la consola de Spring inmediatamente después de esta ejecución.

```
2022-07-06 19:13:39.708  INFO 8260 --- [           main] o.a.c.c.C.[.[.[
2022-07-06 19:13:39.708  INFO 8260 --- [           main] w.s.c.ServletWe
2022-07-06 19:13:40.480  INFO 8260 --- [           main] o.s.b.w.embedde
2022-07-06 19:13:40.496  INFO 8260 --- [           main] oSpringBootCore
2022-07-06 19:15:18.939  INFO 8260 --- [nio-8080-exec-2] o.a.c.c.C.[.[.[
2022-07-06 19:15:18.939  INFO 8260 --- [nio-8080-exec-2] o.s.web.servlet
2022-07-06 19:15:18.940  INFO 8260 --- [nio-8080-exec-2] o.s.web.servlet
Se esta construyendo un objeto de la clase ProductosServiceImpl.
```

Como se puede observar, ahora sí, se muestra la traza que indica que el bean "*ProductoServiceImpl"* se está creando y lo hace en momento de ejecución.

4.5 Capa Services - Configuración de propiedades

En esta sección veremos cómo configurar propiedades globales y recuperarlas en nuestros beans.

4.5.1 Propiedades globales y externalización

Toda aplicación bien formada, independientemente de la tecnología con la cual se encuentre desarrollada, deberá mantener ciertos parámetros configurables fuera del contexto del código fuente. Este procedimiento es conocido como “externalización de parámetros”.

La externalización de parámetros produce varios beneficios sobre nuestro sistema. En principio, unificar todos nuestros parámetros permite una manipulación nítida y fácil de modificar para el administrador del sistema. En nuestro caso unificamos todos estos parámetros en el archivo *“application.properties”*. No obstante existen algunas otras estrategias que proponen la creación de varios archivos de configuración externos que mantienen un “set” de propiedades reunidas de acuerdo a algún propósito en común.

Y en segundo término y tal vez más importante, externalizar estos parámetros en un archivo externo a nuestro código fuente nos brinda la flexibilidad de cambiar valores de configuración “en caliente”. Y con esto nos referimos a cambiar algún valor durante la ejecución del programa sin necesidad de realizar un reinicio o nuevo despliegue en producción.

4.5.2 Configuración de parámetros en Spring Boot

Ahora veremos de forma práctica cómo crear un set de parámetros globales de nuestra aplicación y recuperarlas en nuestro código fuente gracias al uso de ciertas anotaciones. Crearemos cuatro parámetros de tipo global para identificar nuestra aplicación.

Para cargar el valor de estas cuatro claves y utilizarlas en tiempo de ejecución, crearemos un paquete con el nombre *"configurations"* dentro de *"edu.tienda.core"* y dentro del mismo, crearemos una clase java con el nombre *"ConfigurationParameters"* poblado con cuatro atributos que matengan el mismo nombre de las cuatro claves definidas.

```java
package edu.tienda.core.configurations;

import org.springframework.context.annotation.Configuration;

2 usages
@Configuration
public class ConfigurationParameters {

    3 usages
    private String nombre;
    3 usages
    private String pais;
    3 usages
    private String author;
    3 usages
    private String lenguaje;

    public String getNombre() {
        return nombre;
```

También generamos los getter y setters para cada una de las propiedades con el asistente de IntelliJ. Observemos que en la cabecera de la clase, hemos agregado la anotación @Configuration. Esta anotación es de suma importancia ya que su comportamiento es muy parecido a la anotación @Service pero con la diferencia de que estos tipos de beans se inicializan también por Spring de manera prematura pero con distinto propósito.

Los beans anotados con @Configuration, como lo denota la palabra, son clases generadas con el propósito de servir como configuración global de la aplicación. Es decir, en estas clases deberá radicar código fuente que tenga como finalidad ejecutar alguna configuración inicial del sistema como es en este caso.

Podemos observar que cada uno de los atributos tienen el mismo nombre exacto que las claves definidas en el archivo *"configuration.properties"*. Igualmente, todavia esta definición no es totalmente exacta ya que nuestras claves tienen el prefijo *"app"*, como por ejemplo *"app.lenguaje"*. Para emparejar con exactitud estos atributos con las claves correspondientes deberemos agregar una configuración que indique el prefijo de las mismas.

```java
package edu.tienda.core.configurations;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Configuration;

2 usages
@Configuration
@ConfigurationProperties(prefix="app")
public class ConfigurationParameters {
```

Esta anotación indica en primer lugar que Spring deberá cargar el valor de los atributos de esta clase con los valores indicados en el archivo *"application.properties"*. Por supuesto, que Spring podrá realizar esta acción durante el arranque siempre y cuando esos atributos mantengan los mismos nombres que las claves. Y en segundo término, la anotación tiene un atributo *"prefix"* con valor *"app"*, que indica que cada una de estas propiedades tienen un prefijo llamado *"app"* en el archivo de configuración.

Vamos a generar sobre esta clase el método *"toString"* con la asistencia de IntelliJ de la siguiente manera.

```
    @Override
    public String toString() {
        return "ConfigurationParameters{" +
                "nombre='" + nombre + '\'' +
                ", pais='" + pais + '\'' +
                ", author='" + author + '\'' +
                ", lenguaje='" + lenguaje + '\'' +
                '}';
    }
}
```

Para probar la correcta recuperación de estos valores. Vamos a realizar el lookup de este bean dentro del controlador de productos de acuerdo a la siguiente imagen.

```
public class ProductoControllerRest {

    //Se instancia la clase de servicio al estilo "Java Puro"
    1 usage
    @Autowired
    @Lazy
    private ProductoService productosService;

    1 usage
    @Autowired
    private ConfigurationParameters configurationParameters;
```

Observar que el método para inyectar el bean es igual al utilizado para el de *"ProductoService"*. Ahora agregaremos una pequeña traza en el método que recupera los productos dentro de este mismo controlador.

```java
@GetMapping
public ResponseEntity<?> getProductos(){

    System.out.println("params: " + configurationParameters.toString());

    //Se recuperan todos los productos del servicio
    List<Producto> productos = productosService.getProductos();

    //Retornamos los productos del servicio en el body de la respuesta.
    return ResponseEntity.ok(productos);
}
```

Como se puede observar, hemos agregado un *"println"* (más adelante gestionaremos las trazas con un motor de log) al comienzo del cuerpo del método para cuando ejecutemos el endpoint desde POSTMAN se exhiban por consola los valores de esas claves recuperadas desde el archivo *"application.properties"*.

Reiniciamos Spring y ejecutamos con POSTMAN el endpoint para recuperar todos los productos y podremos observar como se muestran esos valores desde la consola.

```
2022-07-06 20:00:12.332  INFO 12496 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on
2022-07-06 20:00:12.345  INFO 12496 --- [           main] oSpringBootCoreDeTiendaOnlineApplication : Started Microservi
2022-07-06 20:00:24.714  INFO 12496 --- [nio-8080-exec-3] o.a.c.c.C.[.[.[/tienda/api/v1]           : Initializing Sprin
2022-07-06 20:00:24.714  INFO 12496 --- [nio-8080-exec-3] o.s.web.servlet.DispatcherServlet        : Initializing Servl
2022-07-06 20:00:24.717  INFO 12496 --- [nio-8080-exec-3] o.s.web.servlet.DispatcherServlet        : Completed initiali
params: ConfigurationParameters{nombre='Tienda Online', pais='Argentina', author='Rafael Benedettelli', lenguaje='es '}
Se esta construyendo un objeto de la clase ProductosServiceImpl.
```

Esta traza se muestra gracias al método *"toString"* definido dentro de la clase *"ConfigurationParametters"* y sirve como prueba fehaciente de que cada uno de sus atributos fueron cargados con los valores correlativos del archivo de configuración. Entonces, ya los disponemos en nuestro código fuente y podremos manipularlos de acuerdo a nuestros objetivos.

4.5.3 Otros mecanismos

Spring dispone de diversos y variados mecanismos para la carga de parámetros externalizados como incluso, recuperar variables de entorno, versión de JAVA que se ejecuta en el equipo, etc.

4.6 Capa Services - Consumiendo API’s

En esta sección veremos cómo consumir una API externa desde nuestro microservicio.

4.6.1 Consumo de recursos externos

Nuestros proyectos Spring que desarrollaremos no solo ofrecerán unas API's de servicios, sino que también, en muchas ocasiones deberemos obtener información de sistemas internos a la organización, e incluso, sistemas externos.

Todas los microservicios de back-end siempre se nutren de información proveída por plataformas como Bases de datos, gestores de contenidos, repositorios documentales, etc que a su vez en algunos casos, son otros microservicios que ofrecen dicha información a través de sus API's REST que exponen.

Spring Boot ofrece un mecanismo sencillo para consumir estas API's permitiendo digerirlas fácilmente en nuestros objetos java "POJO" (nuestras clases de dominio).

4.6.2 Utilizando RestTemplate para el consumo de servicios

"RestTemplate" es la clase más utilizada para consumir servicios externos en Spring Boot. Aprenderemos a utilizarla, pero antes vamos a conformar el contexto.

Antes que todo necesitamos "inventar" esa plataforma externa que vamos a consumir. Para nuestro caso, haremos una API REST al modo "fake" o "dummy" para poder llevar a cabo nuestra tarea.

1. **API Rest Fake.**

En nuestro mismo microservicio y más precisamente en la clase *"ProductoControllerRest"* vamos a generar un nuevo endpoint simulando que es una API de un sistema externo para luego consumirla. Entonces, situados en la clase *"ProductoControllerRest"* creamos el nuevo método que simulara ser una api externa de la siguiente manera.

```
@GetMapping("/fake-productos")
public ResponseEntity<?> fakeProductosAPI(){

    List<Producto> productos = new ArrayList<>(Arrays.asList(
            new Producto( id: 1, nombre: "Camiseta Juventus", precio: 1200.0, stock: 4),
            new Producto( id: 2, nombre: "Camiseta River Plate", precio: 1000.0, stock: 8),
            new Producto( id: 3, nombre: "Camiseta Boca Juniors", precio: 900.0, stock: 1)
            )
    );

    //Retornamos los productos del servicio en el body de la respuesta.
    return ResponseEntity.ok(productos);
}
```

Como se puede apreciar, hemos creado un nuevo método que retorna una lista "dummy" de productos. A este endpoint le hemos asignado la signatura *"/fake-productos"*.

Ahora, solo queda crear el servicio que la consuma. Reiterando lo explicado anteriormente, y con fines prácticos y pedagógicos hemos creado una api como si fuera de un servicio externo a este microservicio pero que en realidad está definido en él mismo.

2. **Consumiendo la api "fake"**

Nos situamos en el paquete *"edu.tienda.core.services"* y crearemos un nuevo servicio que implemente la interface *"ProductoService"* con el nombre *"ProductosServiceImplApiExterna"* y codificaremos el método *"getProductos"* de acuerdo a la siguiente imagen.

```
package edu.tienda.core.services;

import edu.tienda.core.domain.Producto;
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

import java.util.List;
```

```java
@Service("API")
@ConditionalOnProperty(
        value="productos.estrategia",
        havingValue = "EN_OTRA_API")
public class ProductosServiceImplApiExterna implements ProductoService{


    1 usage
    @Override
    public List<Producto> getProductos() {

        RestTemplate restTemplate = new RestTemplate();

        ResponseEntity<List<Producto>> response = restTemplate.
                exchange( url: "http://localhost:8080/tienda/api/v1/productos/fake-productos",
                HttpMethod.GET,  requestEntity: null, new ParameterizedTypeReference<List<Producto>>() {
                });

        List<Producto> productos = response.getBody();


        return productos;
    }
}
```

Pasaremos a explicar el anterior código fuente. En principio hemos creado un servicio alternativo como lo habíamos hecho anteriormente pero que en este caso, la fórmula para retornar los productos, será desde una API externa (que en realidad es una API fake montada en nuestro propio microservicio)

Observar que hemos utilizado las anotaciones @Service y @ConditionalOnProperty para que el nuevo bean se inyecte cuando la clave *"productos.estrategia"* tenga el valor *"EN_OTRA_API"*.

Por lo tanto, para que este bean sea el elegido por Spring entre los otros tres que implementan la misma interfaz, deberemos setear en el *"application.properties"* el valor *"EN_OTRA_API"* dentro de la clave "*productos.estrategia"*.

ctoControllerRest.java | ProductosServiceImplApiExterna.java | application.properties

```
server.servlet.context-path=/tienda/api/v1
server.port=8080

productos.estrategia=EN_OTRA_API
```

Retomando el código fuente que nos interesa.

```java
@Service("API")
@ConditionalOnProperty(
        value="productos.estrategia",
        havingValue = "EN_OTRA_API")
public class ProductosServiceImplApiExterna implements ProductoService{


    1 usage
    @Override
    public List<Producto> getProductos() {

        RestTemplate restTemplate = new RestTemplate();

        ResponseEntity<List<Producto>> response = restTemplate.
                exchange( url: "http://localhost:8080/tienda/api/v1/productos/fake-productos",
                HttpMethod.GET,  requestEntity: null, new ParameterizedTypeReference<List<Producto>>() {
                });

        List<Producto> productos = response.getBody();


        return productos;
    }
}
```

Podemos observar que hemos creado un objeto de la clase *"RestTemplate"* en la primera línea. Luego ejecutamos el método *"exchange"* que sirve para consumir el contenido de una URL (Aclaramos que puede ser cualquier tipo de contenido como HTML, XML, Texto Plano, etc), que en este caso es de tipo JSON. La URL que pasamos es la de nuestra "API Fake".

Como nuestra "API Fake" retorna un array JSON con todos los productos, podemos definir en el segundo parámetro que lo que esperamos es una lista de productos. Spring se encargará de deserializar ese array JSON en una *"List"* de *"Producto"* Java. Y lo hará por lo bajo con su librería Jackson. Para nosotros es transparente y no debemos de lidiar con ese tipo de transformación.

Pero si es importante tener en cuenta que el serializador tratará de emparejar las claves JSON devueltas con los correspondientes atributos java de la clase "*Producto*". Para aquellos atributos que no encuentre coincidencia de nombres, simplemente los omitirá.

Por otro lado, es importante anticipar qué tipo de dato devolverá el JSON. Que en este caso es un Array JSON, es por eso que definimos que nuestra devolución será una "List". Pero si el JSON retornado fuera un objeto JSON como por ejemplo *{"productos":[….]}*. En este último caso, el deserializador fallaría porque esperaría como devolución un Array JSON y no un objeto que lo envuelva. En ese tipo de escenario, deberíamos generar la simetría desde java creando una clase *"ProductosList"* que contenga una lista de productos.

3. Todo listo para probar. Reiniciamos Spring y consumimos el tradicional endpoint

http://localhost:8080/tienda/api/v1/productos

Podemos ver que se obtuvo la lista de productos que estaba definida en el servicio “fake”.

4.7 Capa Services - Lombok

En esta sección veremos cómo omitir código fuente repetitivo con Lombok.

4.7.1 Clases de dominio

Todos nuestros proyectos disponen de un paquete con clases conocidas como “de dominio”. Estas clases Java, también suelen ser denominadas como el “modelo” de nuestro sistema ya que definen las entidades principales de nuestro diseño.

Supongamos que construimos un juego de fútbol manager. En este caso lo primero que definimos, antes de codificar una línea de código, sería las entidades principales que conformarán nuestro juego, como el jugador, equipo, estadio, réferi, director técnico, partido, resultado, fecha, torneo, etc.

Estas tipo de clases son de índole estructural y generalmente no poseen mucha lógica de negocio. Sino que más bien son arquetipos para definir la información de una entidad y nos darán la posibilidad de transportar dicha información entre las distintas capas de nuestra aplicación.

Usualmente, se definirá un Diagrama de Clases siguiendo el estándar UML que nos permitirá identificar estas entidades, pero sobre todo, las relaciones entre ellas. Por ejemplo, un equipo “tiene muchos” jugadores (relación uno a muchos) o un partido “se juega en un” estadio (relación uno a uno).

Para nuestro sistema de compras online hemos definido dos clases de dominio, “Cliente” y “Producto”. Echémosle un vistazo nuevamente a la clase “Producto”.

```java
public class Producto {

    3 usages
    private Integer id;
    3 usages
    private String nombre;
    3 usages
    private Double precio;
    3 usages
    private Integer stock;


    public Producto(){

    }

    6 usages
    public Producto(Integer id,String nombre, Double precio, Integer stock) {
        this.id = id;
        this.nombre = nombre;
        this.precio = precio;
        this.stock = stock;
    }
```

En la parte superior de la clase podemos identificar dos estructuras básicas. La primera es el conjunto de atributos de clase *(id, nombre, precio y stock)*. Luego se identifican dos constructores, uno que es el *"default no-args"* y otro el completo que requiere de todos los atributos para la construcción del objeto.

En la parte inferior de la clase se pueden observar todos los métodos "getter" y "setter" para atender la accesibilidad de sus respectivos atributos, y poder así, respetar el principio de encapsulamiento y privacidad.

```java
public Integer getId() { return id; }

public void setId(Integer id) { this.id = id; }

public String getNombre() { return nombre; }

public void setNombre(String nombre) { this.nombre = nombre; }

public Double getPrecio() { return precio; }

public void setPrecio(Double precio) { this.precio = precio; }

public Integer getStock() { return stock; }

public void setStock(Integer stock) { this.stock = stock; }
```

Como se puede ver, este tipo de clase y esta estructura de código fuente es sumamente frecuente. Es decir, un objeto de dominio sea cual fuera, va a seguir este tipo de patrón de código. En la mayoría de las ocasiones contendrá la lista de atributos, los métodos “getter” y “setters”. Y también los constructores.

Es muy usual, que este tipo de clases implementen también el método *“toString”* que nos permitirá mostrar una representación visual por consola del estado del objeto. Es decir, de los valores de cada uno de sus atributos. Otra función común que puede implementar es *“equals”* para identificar de manera unívoca los objetos mediante algún atributo o combinación de estos.

Este tipo de código es repetitivo. Siempre crearemos la clase con sus atributos, constructores y métodos de acceso. Es por eso que la librería externa Lombok ha ganado mucha adopción, ya que propone una fórmula bastante práctica para omitir la codificación de este tipo de código predecible y repetitivo.

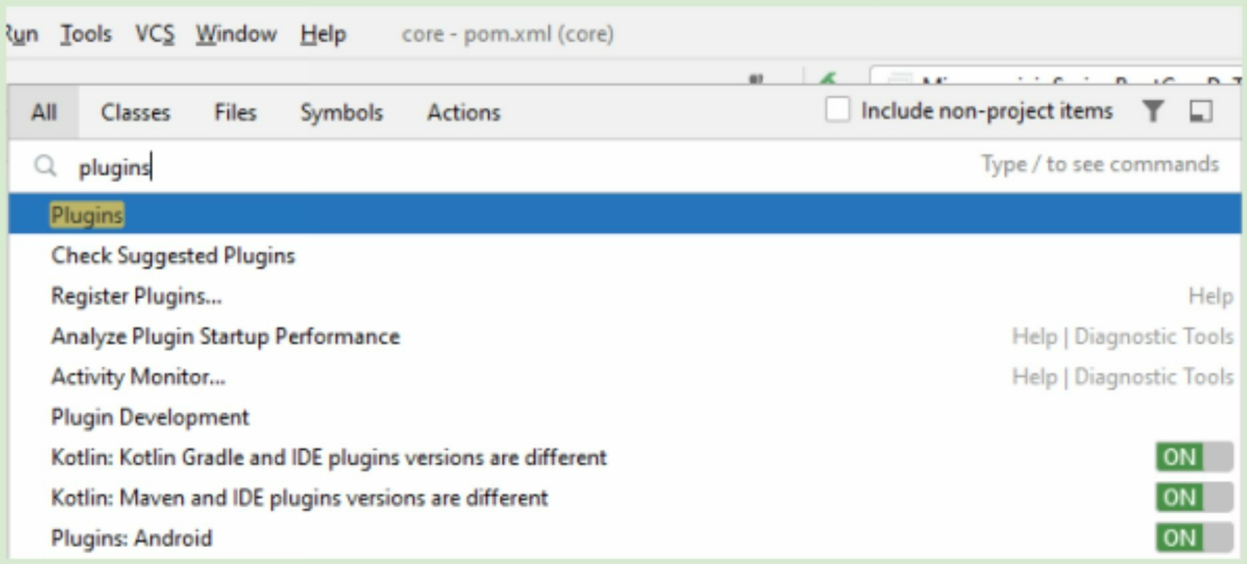
4.7.2 Lombok - Puesta a punto

A continuación incorporaremos Lombok a nuestro proyecto. Primero debemos ingresar la dependencia en nuestro archivo maven del proyecto *"pom.xml"*.

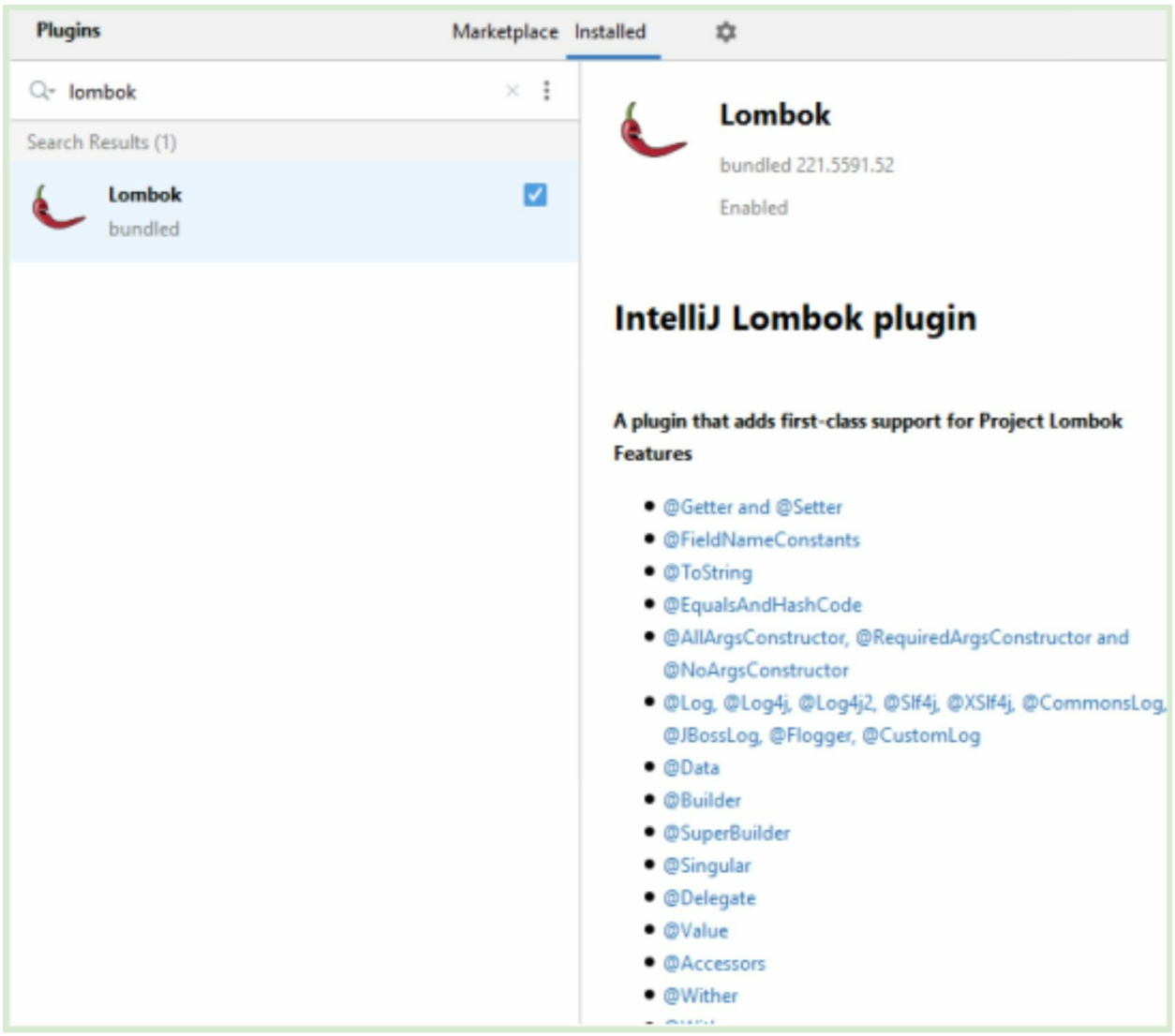

```
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.24</version>
        <scope>provided</scope>
    </dependency>

</dependencies>
```

Una vez ingresada la dependencia, hacer click en el icono que muestra una "m" para que Maven descargue las librerías necesarias desde el repositorio central. Pero con esto no es suficiente. Como Lombok actúa básicamente en tiempo de desarrollo y no de ejecución, ya que es una librería que tiene como propósito ahorrar líneas de código fuente. Debemos soportarnos con un Plugin de IntelliJ.

Para descargar el Plugin de Lombok. En IntelliJ hacer doble click en la tecla "shift" y escribir "plugins".



Ingresar en la opción “plugins” de la lista resultante. En la siguiente pantalla, ingresar en el campo de texto “Lombok” y nos listará en primer lugar el plugin de Lombok con este icono.



Se mostrará el plugin disponible en el marketplace de IntelliJ con un botón “Install”. Hacer click en el botón de instalación y seguir al asistente. Una vez instalado el plugin de Lombok vamos utilizar sus anotaciones propuestas para poder reducir nuestro código fuente de manera significativa.

4.7.2 Lombok - Reduciendo código fuente

Lombok dispone de varias anotaciones java que permite indicar qué tipo de fuente es el que se debería generar en tiempo de compilación.

1. **@Getter y @Setter**

Lo más común y frecuente es que nuestras clases de dominio definen atributos con sus respectivos métodos accesores “getters” y “setters”. Podemos obviarnos la rutina de escribir estos métodos decorando a la clase con las anotaciones @Getter y @Setter de la siguiente manera.

```java
import lombok.Getter;
import lombok.Setter;

25 usages
@Setter
@Getter
public class Producto {

    1 usage
    private Integer id;
    1 usage
    private String nombre;
    1 usage
    private Double precio;
    1 usage
    private Integer stock;


    public Producto(){

    }
```

Ahora sí, podemos borrar todos los métodos “getter” y “setters” ya que Lombok los genera por nosotros en tiempo de compilación gracias a estas dos anotaciones que decoran a la clase en la parte superior de la misma como se puede visualizar en la imagen anterior.

2. @NoArgsConstructor

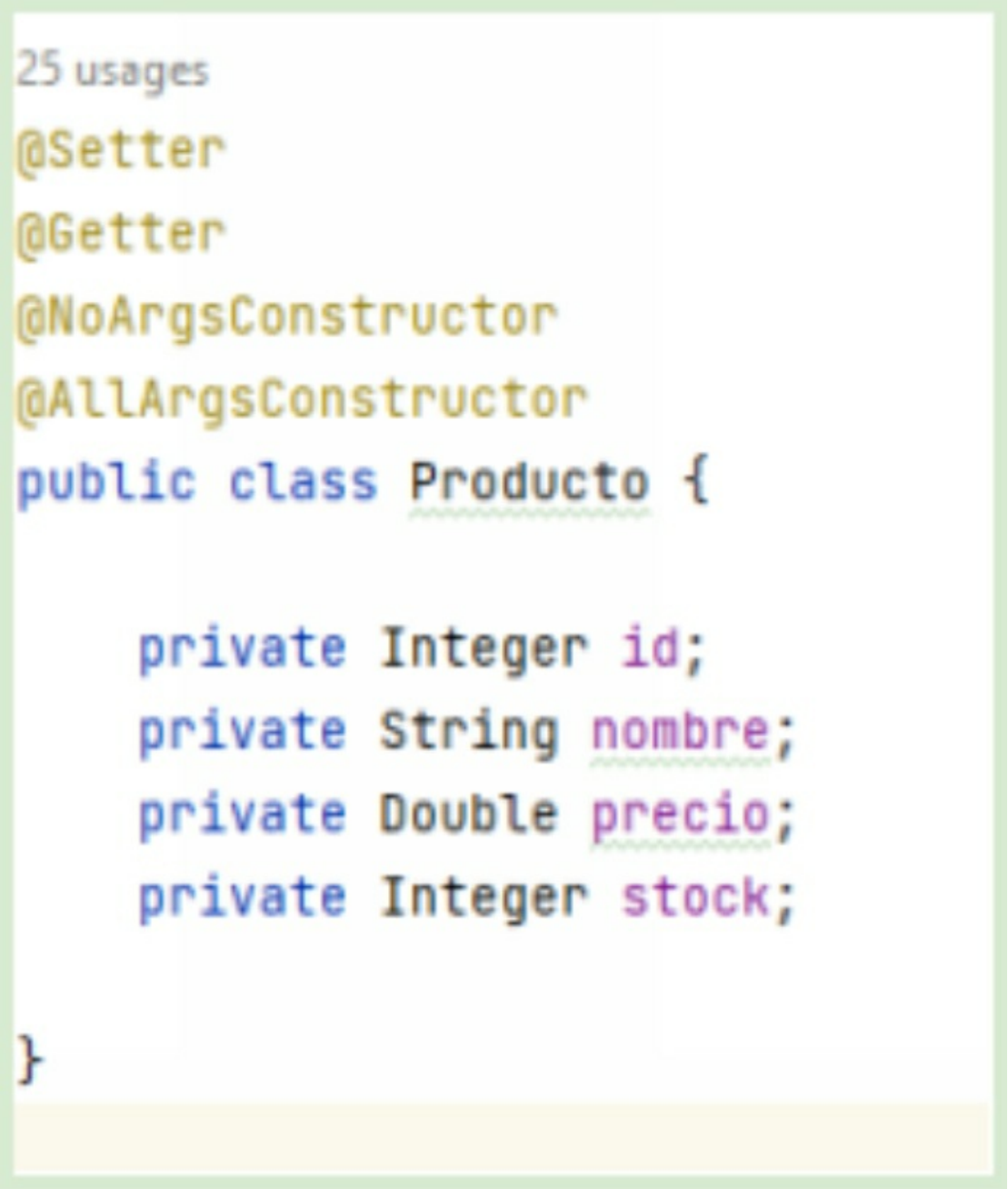
Es muy usual generar explícitamente el constructor “*default*”, que es aquel que no recibe ningún argumento. Incluso, muchas veces es necesario disponer de este constructor ya que será utilizado para instanciar objetos de la clase por librerías externas como por ejemplo Jackson. También podemos obviarnos la codificación del constructor default introduciendo la siguiente anotación.

```java
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

25 usages
@Setter
@Getter
@NoArgsConstructor
public class Producto {

    1 usage
    private Integer id;
    1 usage
    private String nombre;
    1 usage
    private Double precio;
    1 usage
    private Integer stock;

    6 usages
    public Producto(Integer id,String nombre, Double precio, Integer stock) {
        this.id = id;
        this.nombre = nombre;
        this.precio = precio;
        this.stock = stock;
    }
```

Hemos borrado el constructor default ya que definimos la anotación @NoArgsConstructor para que lombok lo codifique por nosotros en tiempo de compilación.

3. @AllArgsConstructor

Por supuesto que también podemos reemplazar el código fuente del constructor completo de argumentos por la anotación @AllArgsConstructor. Nuestra clase va tomando un aspecto mucho más minimalista.

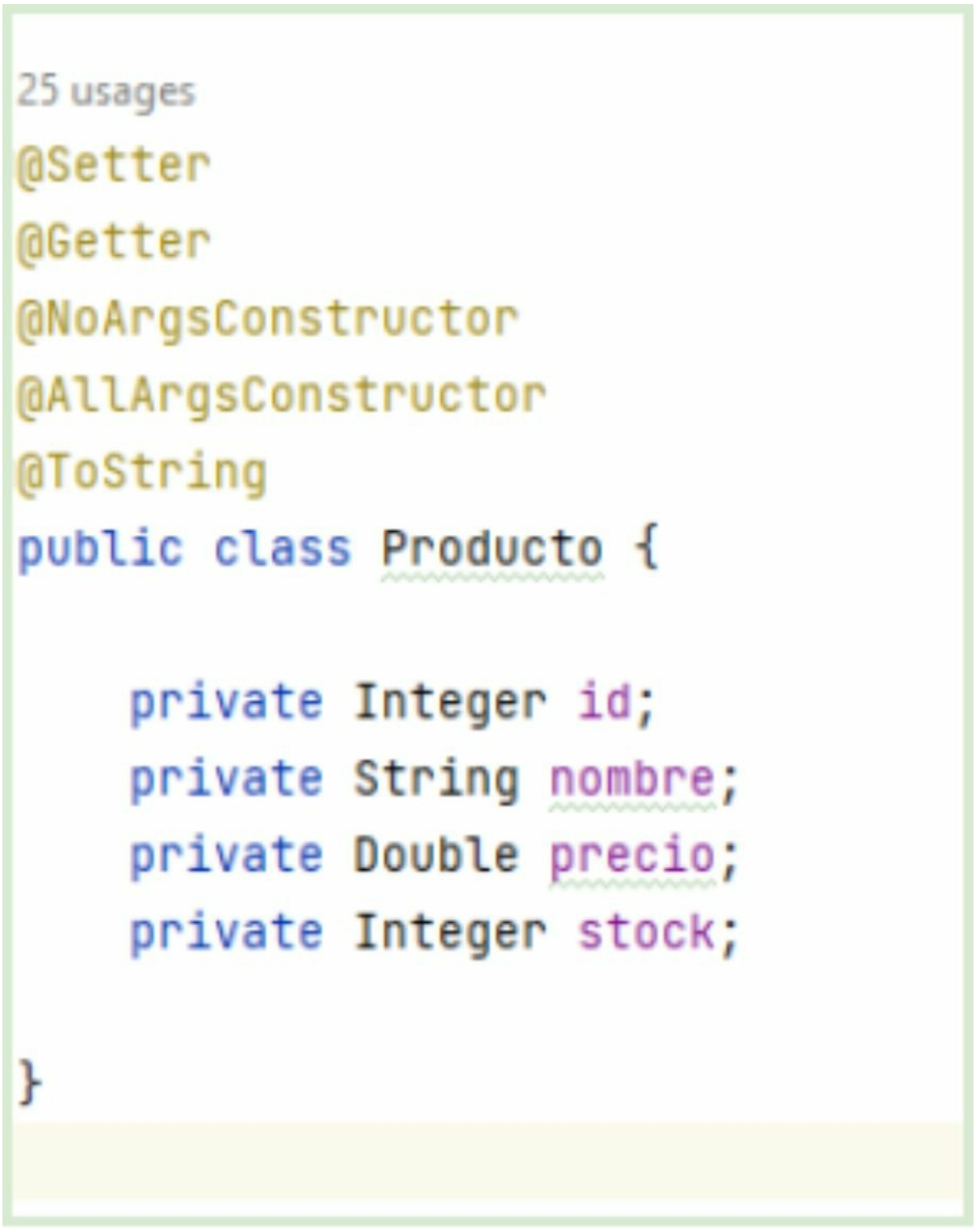


!Una monada de clase!

Hemos reducido unas cuantas líneas de código. No solo nos ahorramos la codificación repetitiva sino también que hemos elevado el aspecto de nuestra clase con la mínima información necesaria. Lombok se encargará del resto.

4. **@ToString**

Ahora agregaremos la anotación @ToString, para que Lombock sobreescriba este método y lo haga como si lo hubiésemos generado con el asistente de IntelliJ.



Lombok nos permite incluso explicitar en la anotación @ToString cual de esos atributos deben mostrarse. En el caso nuestro, se generará el método *"toString"* con todos los atributos de la clase que es lo que queremos.

5. **@EqualsAndHashCode**

En caso de que usemos nuestros objetos de clase para listas de tipo *“Set” o “Hash”* donde queremos asegurarnos que no se repitan dos ejemplares iguales, es necesario escribir los métodos *“equals”* y *“hashcode”*, que juntos conformarán un contrato que garantice la unicidad de los objetos. Lombok también puede hacer este trabajo, que les aseguro que no es trivial.

Así luce nuestra clase que ahora dispondrá de estos dos nuevos métodos.

6. **@Data**

Como se observa, la clase fue perdiendo en código Java pero ganando en anotaciones. Lombok dispone de una anotación “mágica” llamada @Data que permite resumir y contener casi todas estas anotaciones en una sola. Aquí está la nueva clase java.

```java
package edu.tienda.core.domain;

import lombok.*;

23 usages
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Producto {

    private Integer id;
    private String nombre;
    private Double precio;
    private Integer stock;
}
```

Ahora sí que tenemos una clase sumamente “minimizada”. @Data incluye todas las anteriores anotaciones excepto la de constructores. Por lo tanto, de ahora en adelante, lo más

práctico es decorar estas clases con @Data.

Lombok dispone de muchas más anotaciones, que en algunos casos no son las más frecuentes y son de índole más avanzada. En este apartado hemos tratado de mostrar aquellas que son las esenciales e indispensables.

f. @Slf4j

Ahora nos apartamos un poco de las clases estructurales de dominio para situarnos nuevamente en las clases de servicio. Lombok no solo brinda cobertura en las simples clases de modelo, sino también dispone de algunas anotaciones que son muy útiles para otras áreas del proyecto.

Hasta ahora en nuestro libro hemos escrito trazas por consola utilizando la rudimentaria *"System.out.println"*. Esto lo hemos hecho para no desviarnos de nuestros temas principales.

Lo recomendado es utilizar un framework de trazas o logging. Spring incorpora de forma automática y transparente a nosotros el framework Logback. Logback ha demostrado disponer de una de las cualidades que más se le demanda a este tipo de librerías: "Performance".

Un código fuente común y repetitivo es el que construye la instancia singleton de los "loggers". Lombok nos ahorra esta tarea con la anotación de clase @Slf4j

```
1 usage
@Lazy
@Slf4j
@Service("MEMORY")
@ConditionalOnProperty(
        value="productos.estrategia",
        havingValue = "EN_MEMORIA")
public class ProductosServiceImpl implements ProductoService{
```

Como se puede observar hemos decorado la clase con la anotación @Slf4j. Gracias a esta "decoración", disponemos en nuestro código fuente de la variable *"log"* como si nosotros la hubiéramos definido. Vamos a utilizarla para generar algunas trazas en nuestra clase *"ProductosServiceImpl"*. Aprovechamos y cambiamos nuestra línea de *"System.out.pritnln"* por *"log.info"* de esta manera.

```
    public ProductosServiceImpl(){
        log.info("Se esta construyendo un objeto de la clase ProductosServiceImpl.");
    }
```

¿Por qué Slf4j?

Slf4j es un “Facade de loggings”. Intuyo que con esta frase generamos más dudas que antes. Un “Facade de loggings” son una series de clases abstractas que permiten manejar distintos tipos de implementación. En el caso de Slf4j, nos permite manejar de forma abstracta varias implementaciones de logging como Logback, Log4j, July, etc.

La ventaja de utilizar la anotación @Slf4j es que cargará el sistema de trazas definido por default en la solución. Que para nuestro caso es Logback. Pero si el día de mañana quisiéramos importar dependencias de Lo4j2 y definirlo como el principal para nuestro proyecto. No deberíamos cambiar una sola línea de código ya que Slf4j cargará el nuevo sistema de trazas definido para generar los logs necesarios.

Capítulo 5

5.1 Capa Persistencia - Introducción

En este capítulo haremos una introducción a la capa de persistencia.

5.1.1 Persistencia de datos

Estamos de acuerdo que un sistema no sería real sino contara con la posibilidad de guardar datos en algún medio persistente. Imaginémonos un software bancario o de aeropuerto que cada vez que se reinicia pierde la información de vuelos o cuentas cargada anteriormente.

Por esa razón, todos los sistemas vienen acompañados de Bases de datos. Desde el punto de vista de la arquitectura, la sociedad “sistema + base de datos” componen una unidad o entidad unificada. No obstante, desde la radiografía más interna de nuestro microservicio, la base de datos representa un artefacto distinto.

Esto significa que es una plataforma que se ejecuta en otro proceso, escucha en otro puerto, mantiene un código fuente distinto. Es otro programa o sistema en sí. Por lo tanto no conforma parte de nuestro proyecto de desarrollo o código fuente.

Desde esta perspectiva, siempre debemos realizar tareas para preparar nuestro medio de almacenamiento que en lo general son Bases de datos, pero con la evolución de los sistemas corporativos, en algunas oportunidades, estos medios pueden ser Sistemas de gestión documental, Sistemas de gestión de contenido, Bases de datos no relacionales, Archivos en disco o cualquier otra plataforma empresarial.

5.1.2 Puesta a punto de la base de datos

En principio debemos tomar una decisión acerca del motor de Base de datos que queremos seleccionar. Una vez llevada a cabo esta tarea, nos queda la instalación y configuración como cualquier tipo de software nuevo. Finalmente deberemos establecer una conexión desde nuestro artefacto de software.

5.1.3 Base de datos Postgres

En los últimos tiempos hubo una gran adopción de la base de datos Postgres por parte de la comunidad Java. Esta elección se podría explicar gracias a su robustez, fiabilidad y rendimiento. Pero más allá de todos estos beneficios, el real motivo del cambio fue debido a la absorción de MySql por Oracle.

Este “canibalismo empresarial” que atenta contra el buen espíritu del código libre forjó a que la comunidad Java cambie el rumbo hacia otros puertos. La idea era buscar una base de datos de código libre y abierto. Una base de datos Open Source de “pura cepa”.

5.1.4 Docker

Para facilitarnos las tareas, vamos a utilizar la solución “dockerizada” de Postgres. Antes que nada, cabe mencionar que “Docker” ha creado una revolución en materia de desarrollo durante los últimos tiempos. Esto es debido a su flexibilidad que permiten sus contenedores para el despliegue de software auto-contenido y corriendo en un entorno configurado, aislado y exclusivo.

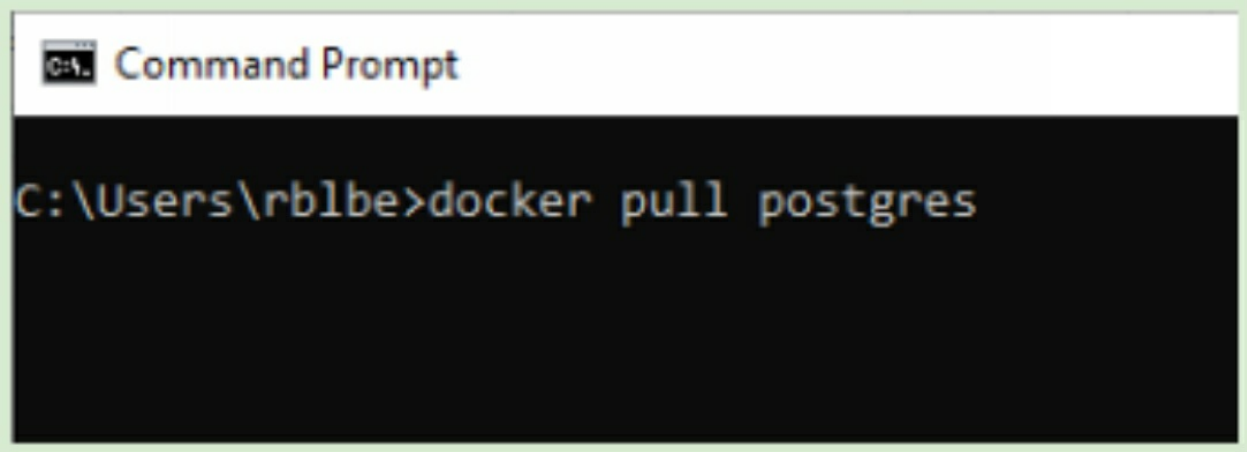
En sí y para explicarlo fácilmente, docker son máquinas virtuales que contienen un sistema operativo con las mínimas librerías del kernel. Uno de los grandes beneficios de docker, es que son contenedores super ligeros con el propósito de ejecutar programas de software en un entorno aislado y configurado exclusivamente para dicho propósito.

Hoy en día, a la hora de instalar cualquier tipo de software, una habitual e inteligente idea es descargar la imagen docker de dicha herramienta. Como por ejemplo lo que haremos ahora, que es descargar la imagen docker de Postgres. Pero también esto vale para otras plataformas como Mongo DB, Alfresco, etc.

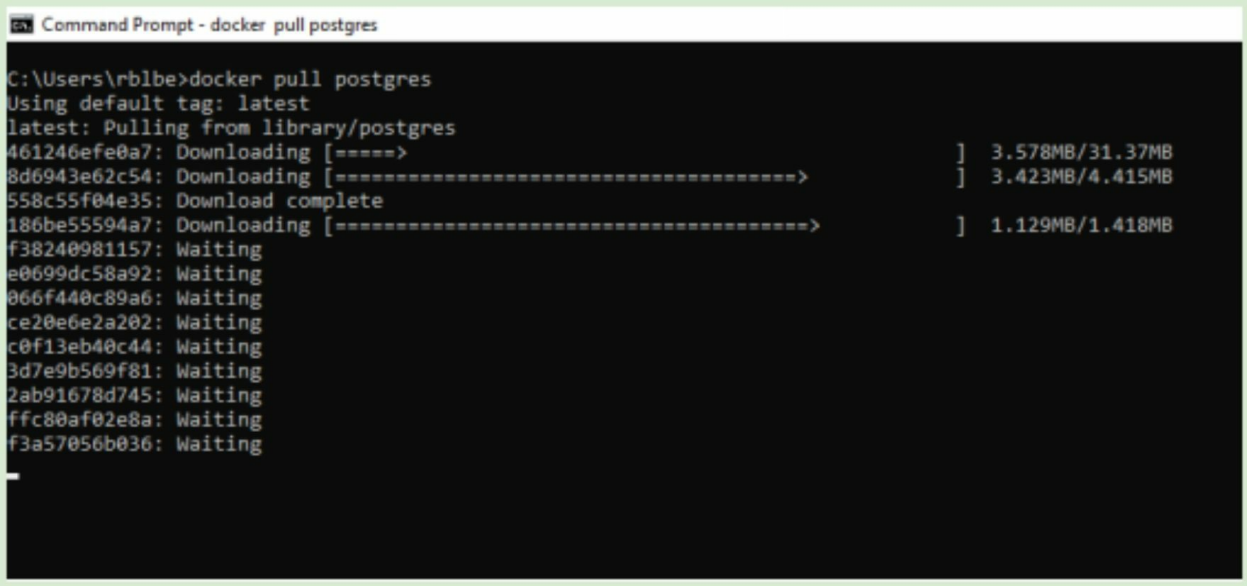
Incluso, nuestro propio proyecto Spring, en un futuro podríamos “dockerizarlo” para disponer su entrega como imagen docker. Desde el sitio https://www.docker.com/get-started/ podemos instalar la versión de docker disponible para nuestro sistema operativo.

5.1.5 Descarga de la imagen Docker de Postgres

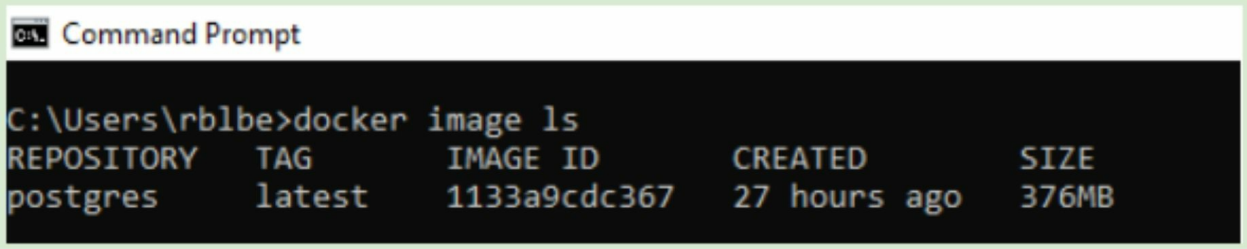
Para descargarnos la imagen docker abriremos una terminal (en este caso en windows pero los siguientes comandos valen igual para linux) e ingresamos el comando *"docker pull postgres"*.



Este comando descarga la última versión de la imagen de Postgres disponible desde el hub central de Docker.



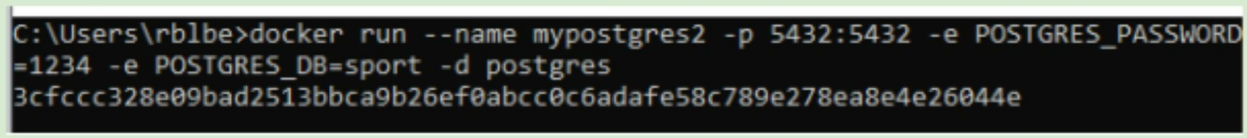
Este proceso demora un tiempo de acuerdo al ancho de banda que tengamos disponible. Al finalizar la descarga podremos comprobar su existencia listando las imágenes docker disponibles en nuestro equipo con el comando *"docker image ls"*



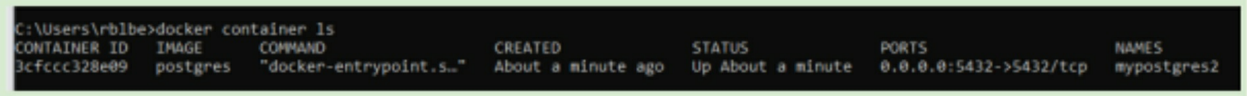
Una imagen Postgres es un esqueleto o molde de la máquina virtual. Mientras que los contenedores son instancias reales corriendo de esa imagen. Para experimentar una analogía con la orientación a objetos, Las imágenes son a las clases como los contenedores son los objetos. Esto significa que podemos correr varias instancias de una misma imagen docker. Para nuestro caso solo vamos a precisar ejecutar una sola con el siguiente comando.

docker run --name mypostgres2 -p 5432:5432 -e POSTGRES_PASSWORD=1234 -e POSTGRES_DB=tienda -d postgres

Si todo sale bien, la consola nos mostrará un identificador largo de esta manera.



Podemos ejecutar el comando *"docker container ls"* para verificar que el contenedor se esté ejecutando.



Como se puede observar, nuestro container está ejecutando y escuchando en el puerto *5432* listo para aceptar conexiones.

Recordemos el comando de Start Up del contenedor:

```
docker run --name mypostgres2 -p 5432:5432 -e
POSTGRES_PASSWORD=1234 -e POSTGRES_DB=tienda -d postgres
```

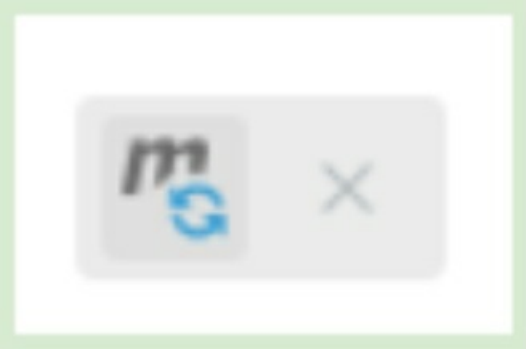
En azul resaltamos dos parámetros que serán necesarios para nuestra conexión desde Spring. Una es el pasword y la otra el nombre de la base de datos que el contenedor de docker creará cuando ejecute el inicio. Solo nos falta el usuario de la base de datos, que por default es *"postgres"*. Ya contamos con todos los datos necesarios para probar nuestra conexión.

5.1.6 Conexión a Postgres desde Spring

En primer lugar debemos incluir dos dependencias a nuestro proyecto. Las situaremos en nuestro archivo *pom.xml* dentro de los tags *<dependencies>....</dependencies>*

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
      </dependency>
      <dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
      </dependency>
```

Una vez adheridas las dos dependencias, deberíamos hacer click en el icono pequeño de maven (se muestra con una “m”) en el ángulo superior derecho de la pantalla.



En la barra de estado inferior del IDE IntelliJ se mostrarán mensajes que indican la descarga de estas nuevas dos dependencias. Una vez finalizada la descarga procederemos a configurar los parámetros de conexión en Spring.

Toda conexión a base de datos para un proyecto Spring Boot debe configurarse en el archivo *“application.properties”*. Lo haremos de la siguiente manera.

```
#Conexion a Base de datos
spring.datasource.platform=postgres
spring.datasource.url=jdbc:postgresql://localhost:5432/tienda2
spring.datasource.username=postgres
spring.datasource.password=1234
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect

#Ejecutar el contenedor Docker.
#docker run --name mypostgres2 -p 5432:5432 -e POSTGRES_PASSWORD=1234 -e POSTGRES_DB=tienda -d postgres
```

Observen que hemos adherido cinco claves fundamentales para la conexión.

En la primera indicamos que la plataforma será *"postgres"*. La segunda y tal vez más importante es la URI del servicio de Base de Datos que incluye el protocolo de comunicación (JDBC) y los datos de host y puerto. Al final se indica el nombre de la base de datos que debe coincidir con el de la línea de ejecución de docker.

Más abajo se indican usuario y password. Y finalmente el dialecto del ORM Hibernate que se utilizara para esta interacción. Además, hemos agregado de forma comentada la instrucción o comando de docker para iniciar el contenedor de Postgres. Esto nos va a ser muy útil cada vez que necesitemos iniciar el contenedor.

Iniciamos Spring Boot y verificamos que en las trazas de consola no ocurra ningún tipo de excepción. Si no se lanza ninguna excepción durante el arranque significa que nos hemos conectado con éxito a nuestra base de datos.

```
Application : Starting MicroservicioSpringBootCoreDeTiendaOnlineAppli
Application : No active profile set, falling back to 1 default profil
ionDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT r
ionDelegate : Finished Spring Data repository scanning in 18 ms. Four
tWebServer  : Tomcat initialized with port(s): 8080 (http)
rdService   : Starting service [Tomcat]
dardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.64]
]           : Initializing Spring embedded WebApplicationContext
tionContext : Root WebApplicationContext: initialization completed in
ource       : HikariPool-1 - Starting...
ource       : HikariPool-1 - Start completed.
.LogHelper  : HHH000204: Processing PersistenceUnitInfo [name: defaul
            : HHH000412: Hibernate ORM core version 5.6.9.Final
n.Version   : HCANN000001: Hibernate Commons Annotations {5.1.2.Final
            : HHH000400: Using dialect: org.hibernate.dialect.Postgre
iator       : HHH000490: Using JtaPlatform implementation: [org.hiber
FactoryBean : Initialized JPA EntityManagerFactory for persistence un
nfiguration : spring.jpa.open-in-view is enabled by default. Therefor
tWebServer  : Tomcat started on port(s): 8080 (http) with context pat
Application : Started MicroservicioSpringBootCoreDeTiendaOnlineApplic
```

Observar que han aparecido nuevas trazas. Estas nuevas, hacen mención a componentes internos destinados para la gestión de base de datos como JPA, Hibernate e Hikari. Por ahora diremos que Hikari es una librería que utiliza Spring para el manejo de pool de conexiones con base de datos. JPA y Hibernate lo explicaremos más adelante.

Por el contrario, si hubiera algún parámetro de conexión incorrecto, Spring arrojaría unas cuantas excepciones por consola. Vamos a forzar la prueba cambiando por ejemplo el nombre de la bd por uno inexistente.

Cambiamos entonces *"tienda"* por *"tienda2"* para forzar el error.

```
spring.datasource.platform=postgres
spring.datasource.url=jdbc:postgresql://localhost:5432/tienda2
spring.datasource.username=postgres
```

Y al ejecutar nuevamente Spring podemos observar varias excepciones por consola.

Como se observa, Spring indica que la base de datos *“tienda2”* no existe, por lo tanto no pudo lograr la conectividad. Lo mismo sucedería de haber un error en el nombre de usuario o contraseña.

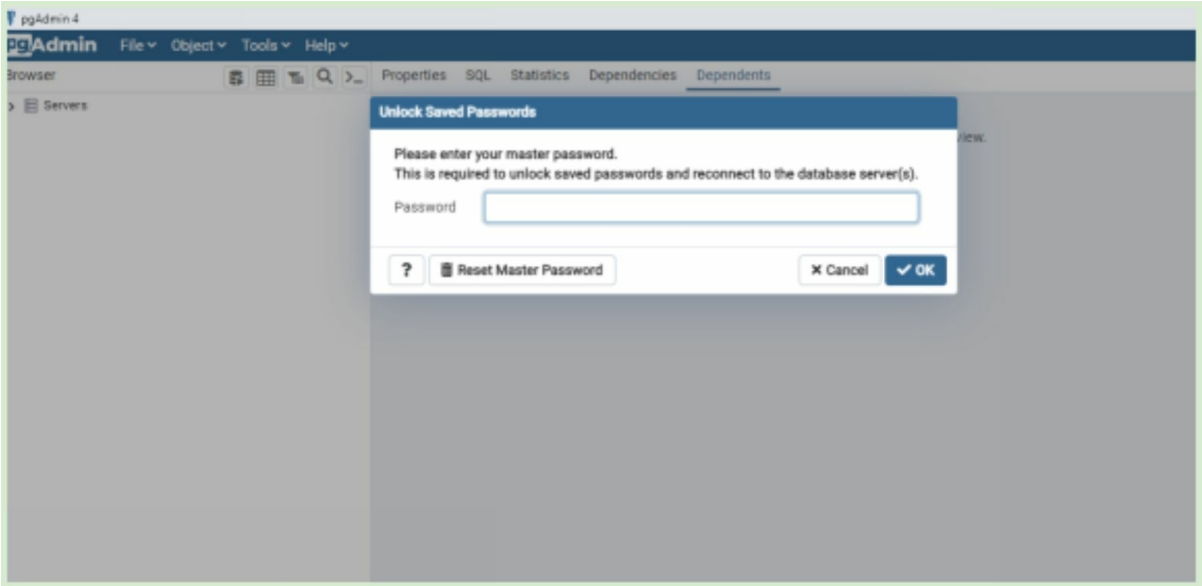
5.1.7 Instalando PgAdmin para administrar Postgres

A continuación instalaremos una herramienta que nos será útil para visualizar los cambios que vayamos realizando en la base de datos Postgres. Su nombre es PgAdmin y hoy en día es de los clientes Postgres más populares. Por supuesto que se trata también de software libre.

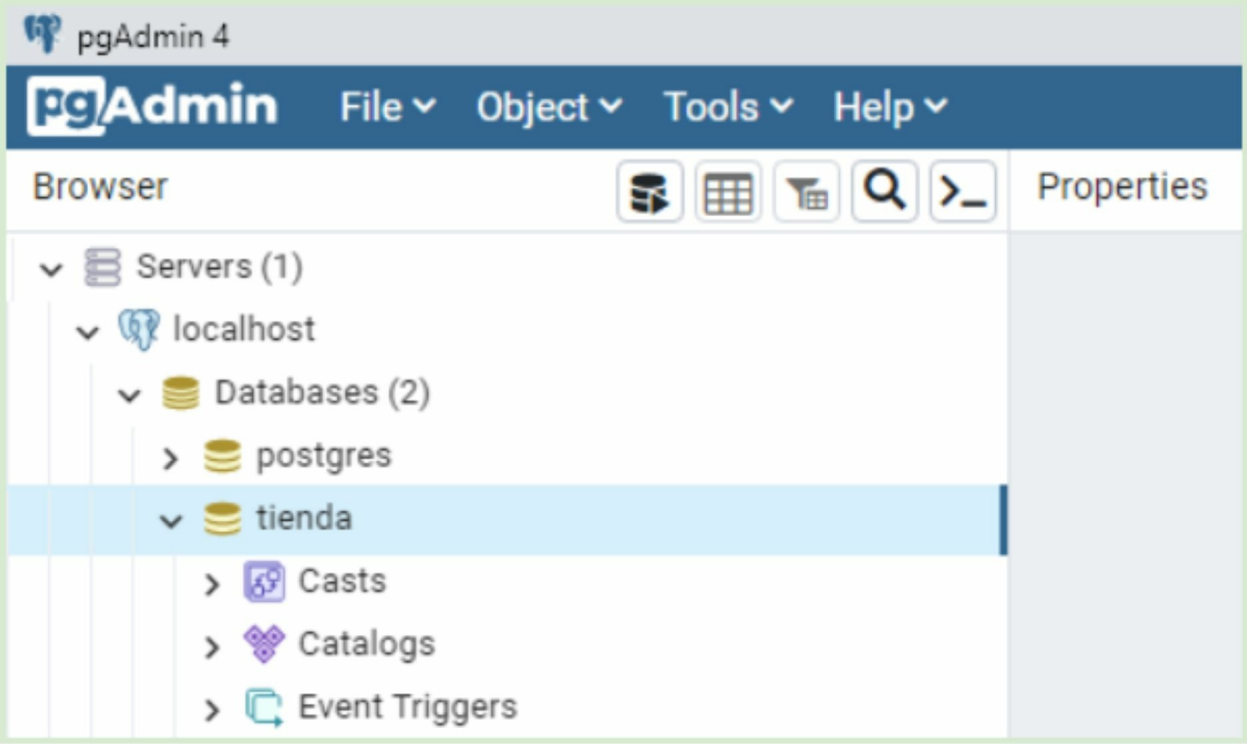
Descargaremos la utilidad para windows desde la siguiente pagina https://www.postgresql.org/ftp/pgadmin/pgadmin4/v6.11/windows/

En este caso y por solo un tema de preferencias, cuando se trata de software de tipo cliente nos inclinaremos por instalar la aplicación directamente y no su imagen docker. En cambio cuando se trate de algún back-end, plataforma o servicio escuchando en algún puerto optariamos por la solución dockerizada. Desde la url mencionada pueden descargar el archivo "exe" que se presenta en el primer enlace. Luego ejecutarlo y seguir el asistente de instalación paso a paso.

La instalación de PgAdmin no requiere ningún tipo de configuración. Es un simple "wizard" que debe completarse paso a paso. Una vez instalado, ejecutarlo y se nos presentará la pantalla de login.



Aquí deberemos ingresar el "master password" y en la siguiente pantalla la clave que hemos configurado para Postgres "dockerizada", que es *"1234"*. Una vez autenticados se nos presentará la siguiente pantalla.



Ingresamos al menú lateral y desplegamos la opción “server - localhost - databases - tienda” y se nos mostrará todas las estructuras de nuestra base de datos.

Por el momento nos detendremos aquí con la herramienta PgAdmin pero en las próximas secciones retomaremos la utilidad para inspeccionar sus tablas a medida que las vayamos creando desde Spring Boot.

5.2 Capa Persistencia - Entidades

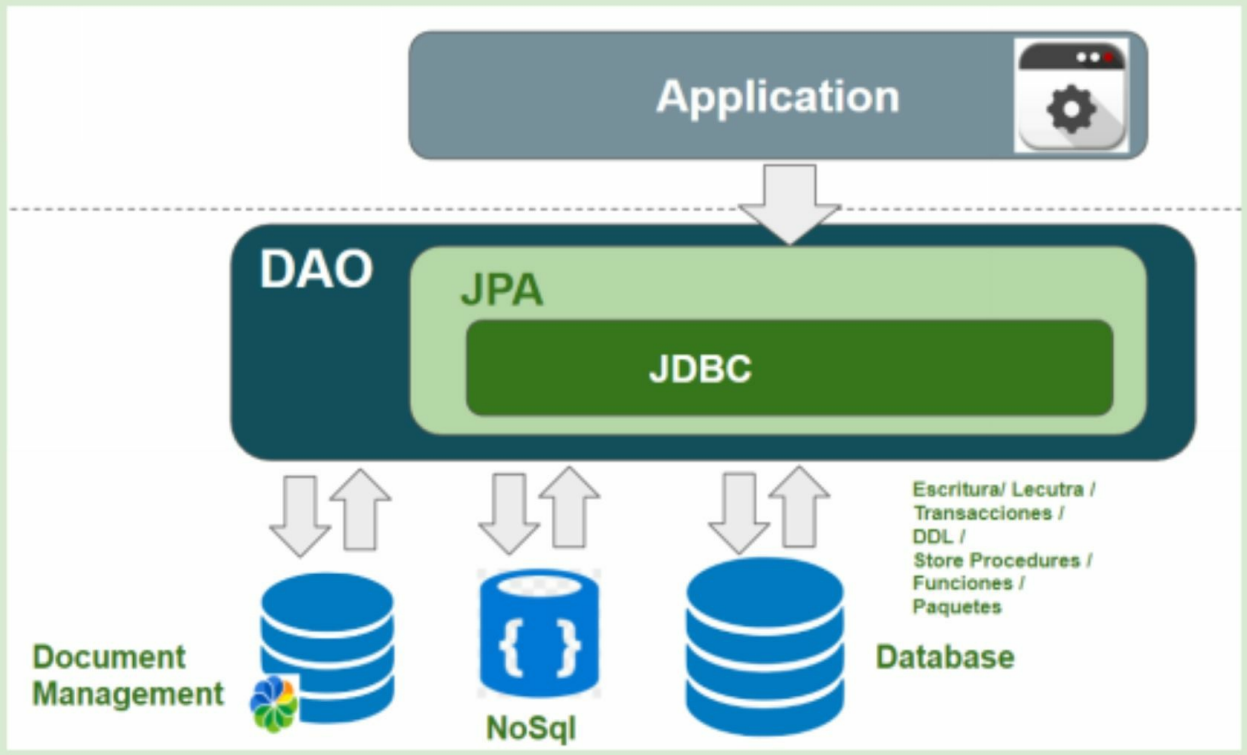
En esta sección crearemos el modelo de entidades JPA.

5.2.1 Java y base de datos

La historia de Java y las bases de datos viene de antaño. Desde las primeras versiones del lenguaje se desarrolló el protocolo JDBC que permitía de una forma bastante práctica interactuar con varios motores de BD.

JDBC nació como un estándar que proponía una serie de interfaces que los fabricantes de base de datos debían implementar. Con el tiempo fueron apareciendo ORM's como Hibernate y JPA entre otros, que permitían mapear el modelo relacional de las bases de datos con los objetos Java. Este mecanismo permite un mayor nivel de abstracción ya que podremos seguir programando en el paradigma orientado a objetos en varias ocasiones salvo en aquellas en las que se necesite ejecutar consultas más complejas.

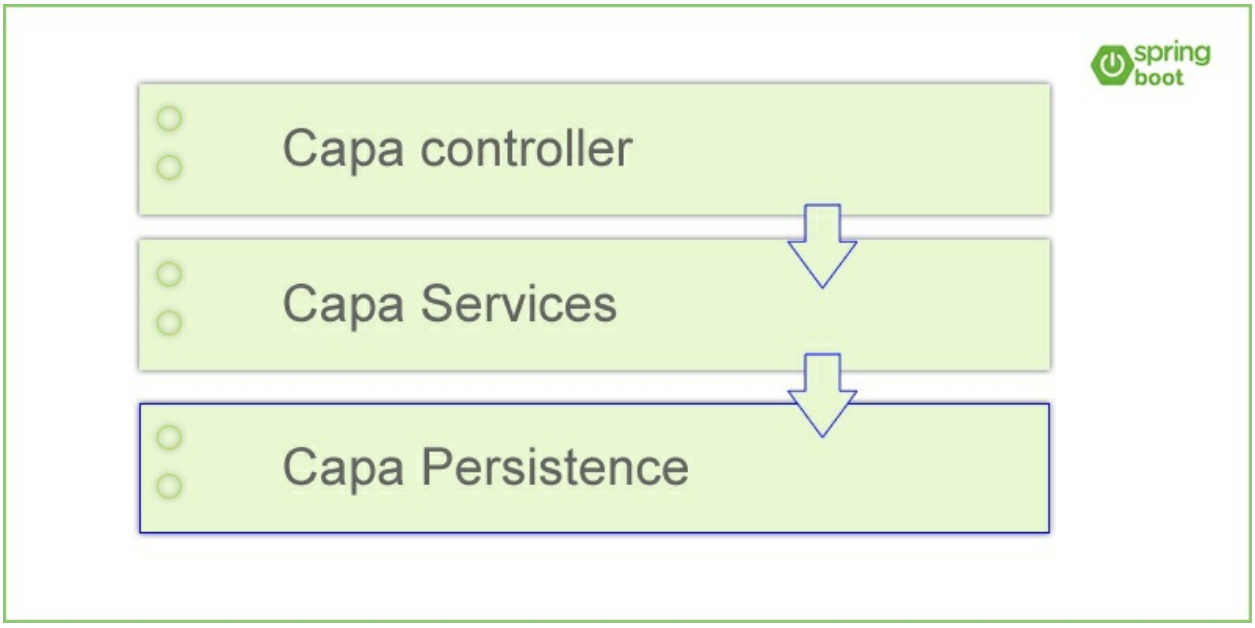
Spring Boot fue un poco más allá y propone una propuesta de consulta y escritura de base de datos mediante la definición de simples interfaces.



Como muestra la infografía, en realidad se trata de un sistema de capas al estilo "envoltorios" donde un estándar alberga a otro.

5.2.2 Capa de persistencia

De aquí en más comenzaremos a programar la capa de persistencia de nuestro proyecto Spring. Como se habrán dado cuenta utilizamos el enfoque de construcción “Top-Down”. Hemos comenzado a construir el microservicio desde la capa superior (“controlador”), pasando por la capa intermedia de servicios (“services”) y ahora construiremos la capa inferior de persistencia.

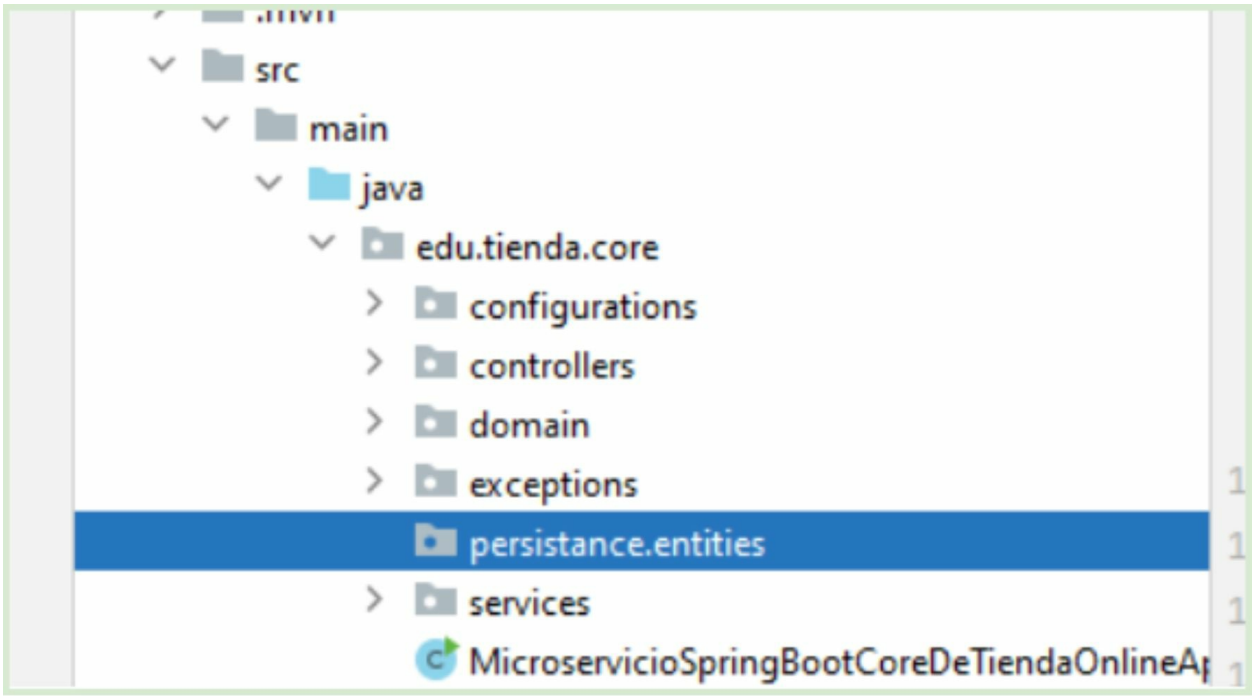


5.2.3 Definición de entidades

Primero debemos analizar qué tablas tendrá nuestra base de datos *"tienda"*. En este momento de acuerdo a la definición de nuestra API Rest podríamos identificar que deberíamos crear una tabla para almacenar los clientes y otra para los productos.

A fines prácticos, nos enfocaremos en los productos únicamente. El estándar JPA nos simplificará bastante la vida en cuanto al manejo de tablas desde nuestro código fuente. Por ahora pensemos en una simple regla: "Por cada tabla crearemos una Entidad JPA".

De acuerdo a esta pequeña regla, vamos a crear en principio un paquete general que se llamará *"persistance"* y agrupará todo el código relacionado con la capa de persistencia. Dentro de este paquete crearemos otro subpaquete llamado *"entities"* donde ubicamos nuestras clases de entidad JPA.



Ahora sí crearemos dentro del paquete *"entities"* nuestra entidad JPA para gestionar la tabla de productos. Crearemos una clase con el nombre *"ProductoEntity"*.

```java
package edu.tienda.core.persistance.entities

public class ProductoEntity {



}
```

Hasta aquí es una clase java común y corriente. Pero desde que la decoramos con la anotación @Entity la transformaremos en una entidad JPA que nos permitirá gestionar la tabla de productos.

```java
package edu.tienda.core.persistance.entities;

import javax.persistence.Entity;

@Entity(name="productos")
public class ProductoEntity {



}
```

Observar que hemos incluido el atributo name en la anotación @Entity especificando cómo será el nombre de la tabla en base de datos. En la base de datos, como bien sabemos, las tablas que representen las entidades de nuestro sistema deberán contener un identificador único. Por eso agregaremos un atributo que representara este ID.

```java
package edu.tienda.core.persistance.entities;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity(name="productos")
public class ProductoEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private Integer id;


}
```

Así como cada clase JPA estará vinculada con una tabla en la base de datos, cada atributo de esta clase estará vinculada con una columna en esa tabla.

En este caso hemos incluido un atributo con identificador *"id"* y de tipo *"Integer"*. Lo hemos anotado con @Id que es bastante intuitiva. Simplemente significa que este atributo será una clave única de la tabla productos. Mientras que la anotación @GenerateValue indica la estrategia con la que la base de datos debería generar ese identificador único. Hay diversas opciones. Para nuestro caso utilizaremos una generación mediante una secuencia que es la más usual en Postgres.

Aclaración: en algunos casos puede ser buena idea definir el atributo *"id"* como *"Long"*, sobre todo si estamos involucrados en un desarrollo que tenga alta concurrencia. Por supuesto que nuestra tabla deberá estar poblada por más columnas para almacenar el precio, nombre y stock del producto. Vamos a crear los restantes atributos. Los podríamos tomar de la clase *"Producto"*.

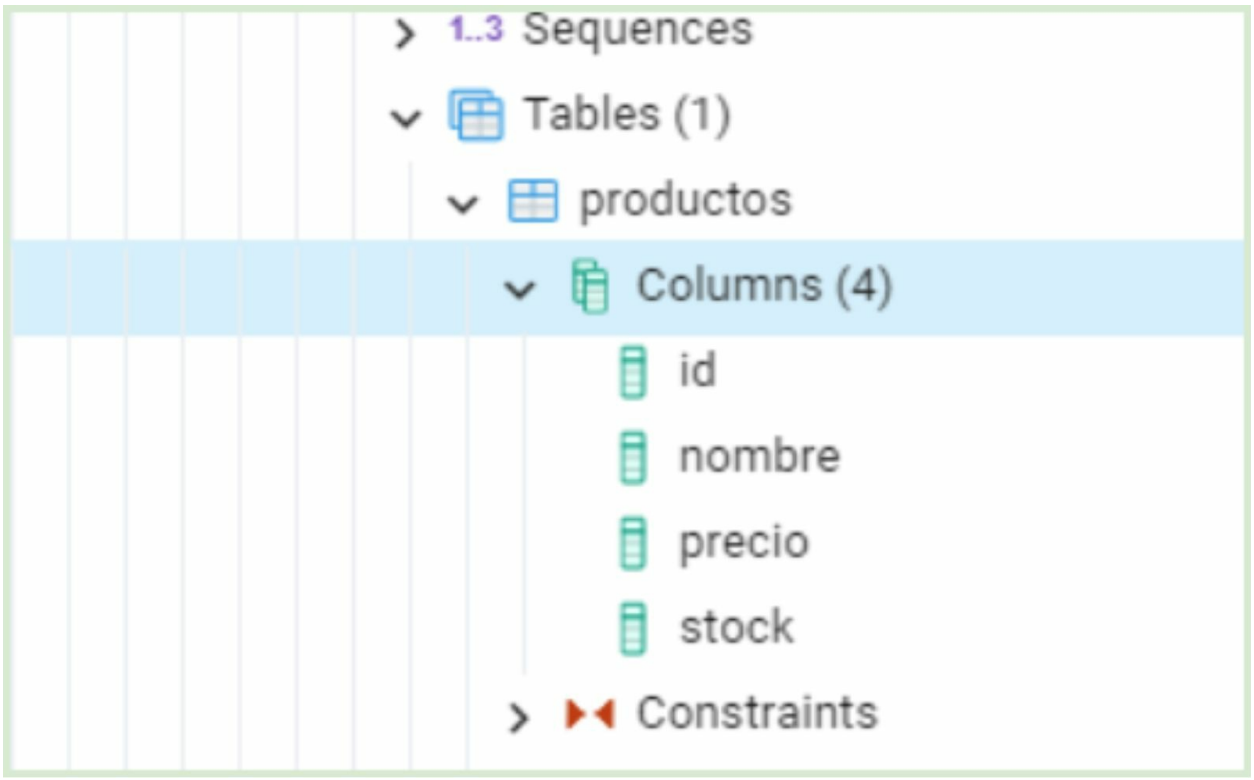
```java
import lombok.Data;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

5 usages
@Entity(name="productos")
@Data
public class ProductoEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private Integer id;
    private String nombre;
    private Double precio;
    private Integer stock;


}
```

No olvidar agregar la anotación de Lombok @Data para que se generen los métodos accesores. Hemos creado nuestra clase JPA para gestionar la tabla de productos.

Con esta definición Spring dispone de la información necesaria para crear la tabla de productos en lugar de que nosotros la definamos "a mano". Para lograr esta automatización debemos incluir un parámetro en el archivo *"application.properties"*.

```
spring.datasource.platform=postgres
spring.datasource.url=jdbc:postgresql://localhost:5432/tienda
spring.datasource.username=postgres
spring.datasource.password=1234
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=update
```

Y al iniciar Spring Boot podremos observar que Spring creará automáticamente la tabla *"productos"* en la base de datos. Lo podremos chequear con la herramienta PgAdmin.



Desplegando la opción de “tables” de nuestra base de datos, ahora encontraremos la nueva tabla “productos” con todas sus columnas, respetando por supuesto, el tipo de dato definido en Java y buscando el más emparejado de los tipos disponibles en Postgres.

El parámetro ***spring.jpa.hibernate.ddl-auto*** permite ser configurado con otros valores de acuerdo a nuestra conveniencia. A continuación, las opciones disponibles y las recomendaciones de cada una según el entorno en el que estemos trabajando.

Valores posibles para spring.jpa.hibernate.ddl-auto:

- **update** - Agrega nuevos datos de esquema pero no elimina nada. (Desarrollo)
- **create-drop** - Elimina el esquema y lo vuelve a crear (Test)
- **validate** - Solo válida y compara el esquema. Si hay inconsistencias lanza una excepcion.
- **none** - Desactiva la generación de esquema (Producción)

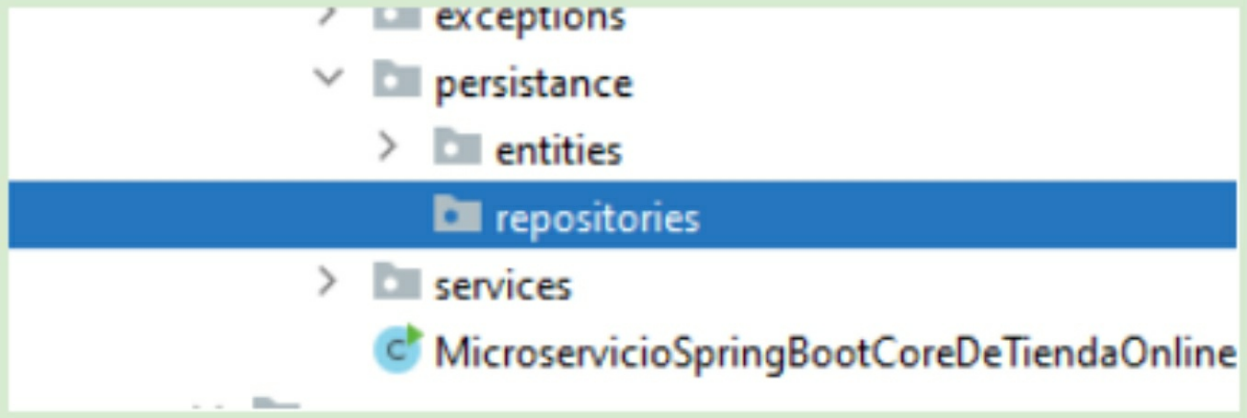
5.3 Capa Persistencia - Repositorios

En esta sección crearemos los repositorios JPA.

5.3.1 JPA Repositories

Ahora que tenemos definida nuestra entidad (JPA Entity) de producto y se encuentra "bindieado" con nuestra tabla productos de base de datos. Vamos a crear una interfaz JPA Repository que nos permite gestionar el alta, baja, modificación y eliminación del producto.

Dentro de nuestro paquete *"edu.tienda.core.persistance"* crearemos un nuevo subpaquete con el nombre *"repositories"* de manera tal que se nos muestre como en la imagen.



A continuación crearemos dentro de este paquete *"repositories"* una interfaz Java con el nombre *"ProductosRepository"*.

```
package edu.tienda.core.persistance.repositories;

public interface ProductosRepository {

}
```

Como siempre decimos, hasta aquí una clase Java como cualquiera hasta que la comencemos a decorar para nutrirla con la potencia de Spring.

```java
import org.springframework.stereotype.Repository;

@Repository
public interface ProductosRepository {

}
```

Esta primera anotación @Repository convierte a esta clase java en un bean de Spring, como si fuera @Service por ejemplo, pero con el propósito exclusivo de interactuar con el esquema de base de datos. Hasta aquí, solo una anotación para que Spring la registre en su contenedor de dependencias y conozca de qué tipo de clase se trata.

Ahora deberemos hacer que esta interfaz extienda de JpaRepository para indicarle que tipo de "Entity" gestionará. Para ello, agregamos los siguientes cambios en la interfaz de acuerdo a la siguiente imagen.

```java
package edu.tienda.core.persistance.repositories;

import edu.tienda.core.persistance.entities.ProductoEntity;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

2 usages
@Repository
public interface ProductosRepository extends JpaRepository<ProductoEntity,Integer> {

}
```

Hemos indicado que la interfaz "*ProdocutosRepository*" extiende de otra interfaz JPARepository, que está disponible en Spring Data y le hemos indicado en los genéricos que va a gestionar la entity "*Producto*" que tendrá un id de tipo "*Integer*".

5.3.2 Creando el CRUD de productos

Volvemos a retomar la capa de servicios para crear uno nuevo que gestione los productos desde base de datos.

Recordemos que tenemos disponibles tres implementaciones distintas para la gestión de productos. *"ProductosServicioImpl"* que los recuperaba en memoria, *"ProductosServiceJSONImpl"* que los recuperaba desde un archivo JSON y *"ProductosServiceImplApiExterna"* que los extraía mediante el consumo de un servicio Rest.

Dicho esto, crearemos un cuarto servicio de productos para que los recupere desde base de datos. Ya tenemos todo dispuesto y organizado para hacerlo con nuestra entidad y repositorio. Solo falta crear nuestro nuevo servicio para que pueda orquestar estos componentes de persistencia.

Crearemos entonces, la clase *"ProductosServiceBDImpl"* en el paquete *"services"*. Por el momento la crearemos con esta base. Para ahorrarnos tiempo, recomiendo hacer una copia de la clase *"ProductosServiceImpl"* aprovechando su parecido.

```java
package edu.tienda.core.services;

import edu.tienda.core.domain.Producto;
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import org.springframework.stereotype.Service;
import java.util.List;

@Service("BD")
@ConditionalOnProperty(
        value="productos.estrategia",
        havingValue = "EN_BD")
public class ProductosServiceBDImpl implements ProductoService{

    1 usage
    public List<Producto> getProductos() {
        return null;
    }

}
```

Nuestra clase luce por el momento con este esqueleto. Como se aprecia, mantuvimos la anotaciones de servicio y condicional para que se cargue cuando el valor de la clave sea *"EN_BD"*. Por ahora el método *"getProductos"* returna *"null"* temporariamente.

Como mencionábamos, la clase *"ProductoServiceBDImpl"* será la encargada en orquestar los componentes del paquete persistencia que hemos definido ("*Producto*Entity" y *"ProductosRepository"*).

Como se trata de un sistema de tres capas superpuestas siempre la capa superior gestionará las clases de la capa inferior. Así, lo hemos hecho también con la capa controladora que inyectaba beans de la capa de servicio para ejecutar sus métodos.

Aquí, haremos lo mismo. La capa de servicios inyectará beans de la capa de persistencia. Vamos a inyectar entonces el bean de persistencia de productos en nuestra clase de servicios.

```java
import edu.tienda.core.domain.Producto;
import edu.tienda.core.persistance.repositories.ProductosRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import org.springframework.stereotype.Service;
import java.util.List;

@Service("BD")
@ConditionalOnProperty(
        value="productos.estrategia",
        havingValue = "EN_BD")
public class ProductosServiceBDImpl implements ProductoService{

    @Autowired
    private ProductosRepository productosRepository;

    1 usage
    public List<Producto> getProductos() {
        return null;
    }

}
```

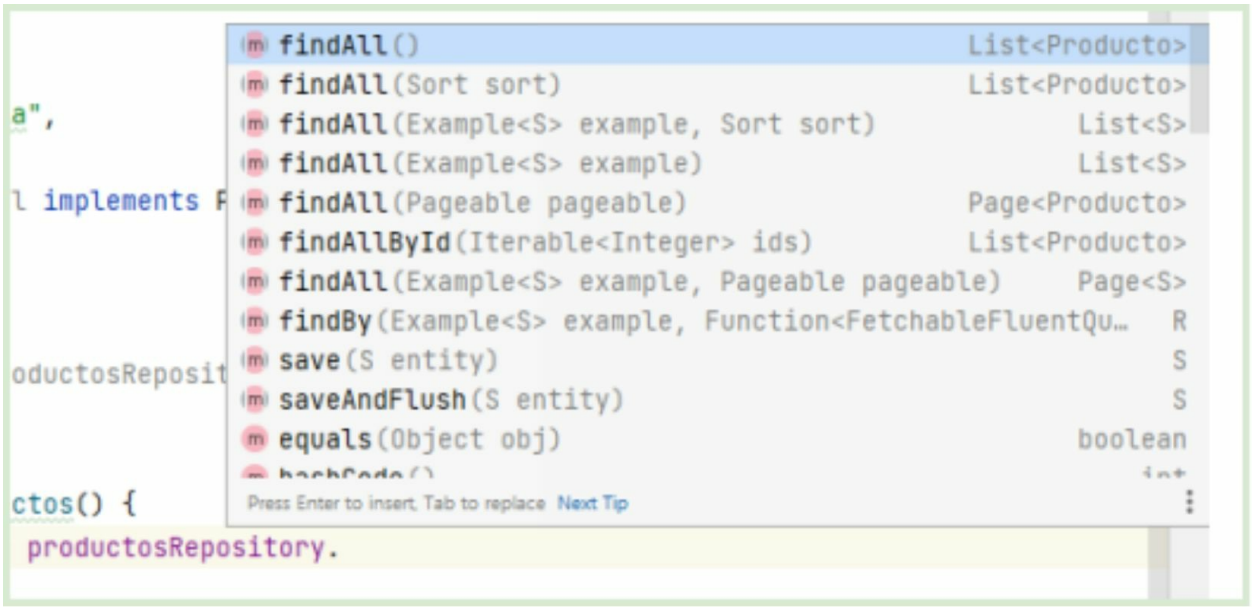
Lo hemos hecho con el mecanismo ya conocido @Autowired. En este caso particular, el lector puede estar preguntándose ¿Qué clase concreta estará inyectando Spring en esta variable de interfaz?

La respuesta es que cuando definimos una interfaz que extiende de *"JpaRespository"*, Spring crea una clase "concreta" en el "Start Up" de la aplicación de forma transparente e invisible a nosotros. Es esa clase concreta y "oculta" que Spring inyectará en este servicio.

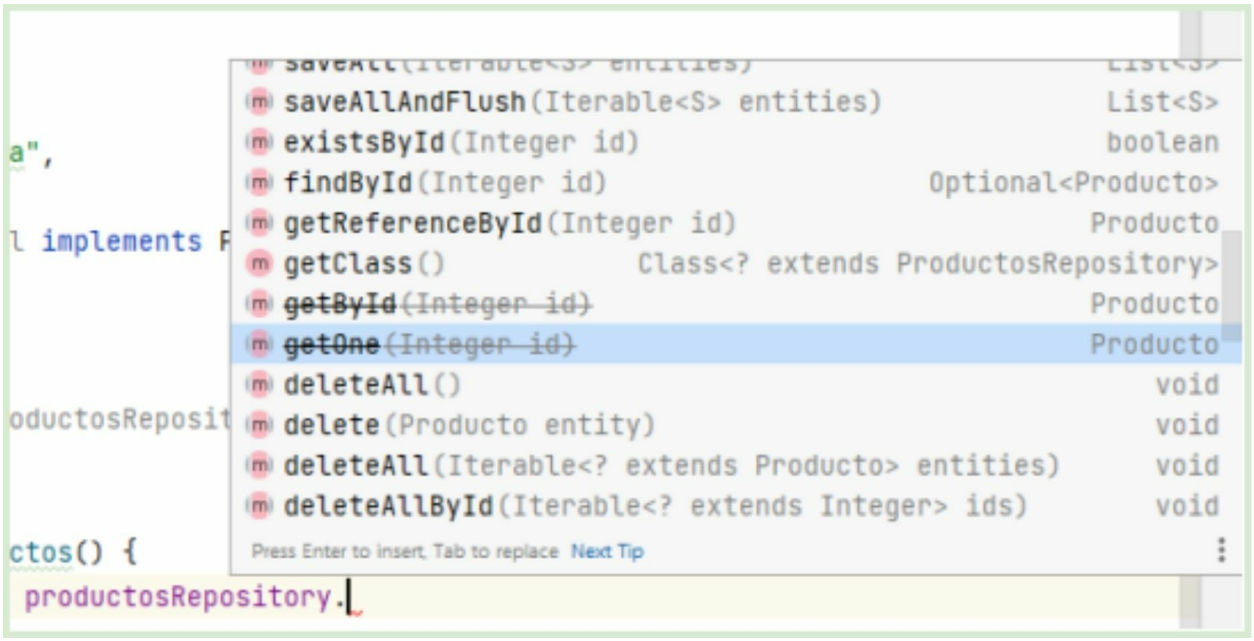
Vamos a usar *"productosRepository"* para recuperar todos los productos desde la base de datos postgres.

```java
1 usage
public List<Producto> getProductos() {
    List<ProductoEntity> productosEntities = productosRepository.findAll();

    return null;
}
```

La implementación de este método todavía es provisoria ya que sigue retornando “null”. Simplemente hemos ejecutado el método *“findAll”* que nos retorna todos los productos de la tabla *“productos”* sin discriminar por ningún tipo de dato. Al escribir punto “.” luego de la variable *“productosRepository”*, IntelliJ nos mostrará un asistente con todos los métodos disponibles. Echémosle un vistazo.



Como pueden observar ya tenemos varios métodos disponibles para las operaciones básicas de base de datos como recupero, recupero por *id*, guardado y también eliminación que se ve en la siguiente imagen.



Apreciemos la cantidad de queries y mapeos que Spring nos ahorra con las interfaces JPA. Y

lo mejor de todo es que es meramente declarativo. No hemos implementado ninguna clase concreta con algoritmos específicos para lidiar con la comunicación a base de datos. ¡Solo hemos declarado una interfaz!

5.3.3 Data Access Object vs Data Transfer Object

Aquí hay que detenernos para observar que el método *"findAll"* nos devuelve todos los productos en objetos de la clase *"ProductoEntity"*. Pero el servicio actual, dentro de este método, está preparado para retornarle al controlador todos los productos como objetos de la clase *"Producto"*.

Tal vez se pregunten, para qué disponemos de dos clases productos que son similares, con mismos atributos, etc. En realidad son iguales pero con propósitos distintos. La clase *"Producto"* pertenece al modelo de datos y tiene el rol de "Data Transfer Object". Esto significa que se utiliza para representar la información del producto en la API Rest. Mientras que la clase *"ProductoEntity"* tiene un rol de tipo "Data Access Object" que tiene como propósito transportar la información desde base de datos hasta la capa de servicio.

Vamos a profundizar en esta separación de clases más adelante ya que conlleva un concepto de diseño bastante importante.

5.3.4 Mapeo manual de entidades a modelo

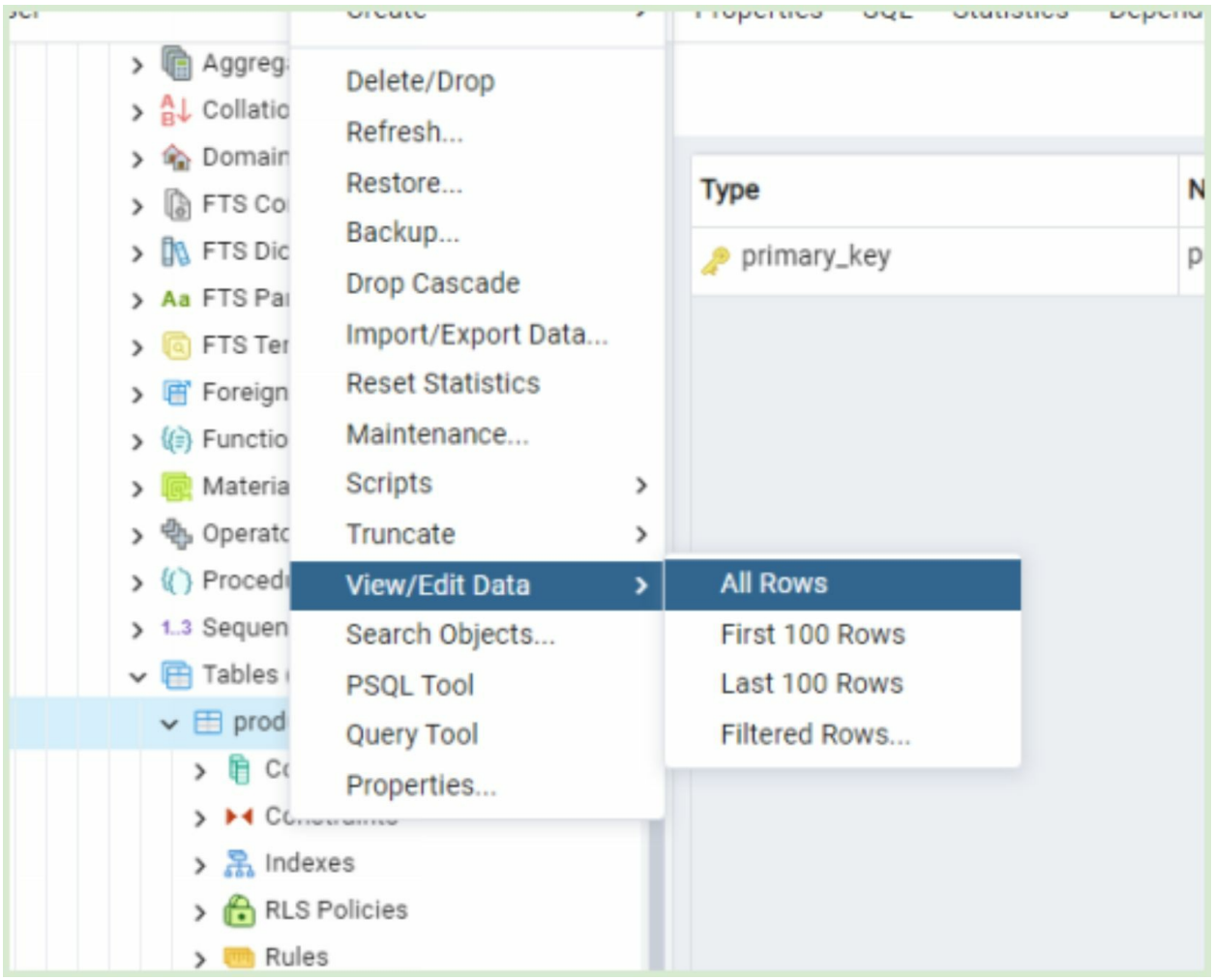
Vamos a realizar un mapeo manual de nuestros objetos de clase *"ProductoEntity"* a objetos de la clase *"Producto"* de la siguiente manera. Primero lo haremos con un código Java convencional y rústico.

```
1 usage
public List<Producto> getProductos() {
    List<ProductoEntity> productosEntities = productosRepository.findAll();
    List<Producto> productos =  new ArrayList<>();

    for (ProductoEntity productEntity : productosEntities){
        Producto producto = new Producto();
        producto.setId(productEntity.getId());
        producto.setNombre(productEntity.getNombre());
        producto.setPrecio(productEntity.getPrecio());
        producto.setStock(productEntity.getStock());
        productos.add(producto);
    }
    return productos;
}
```

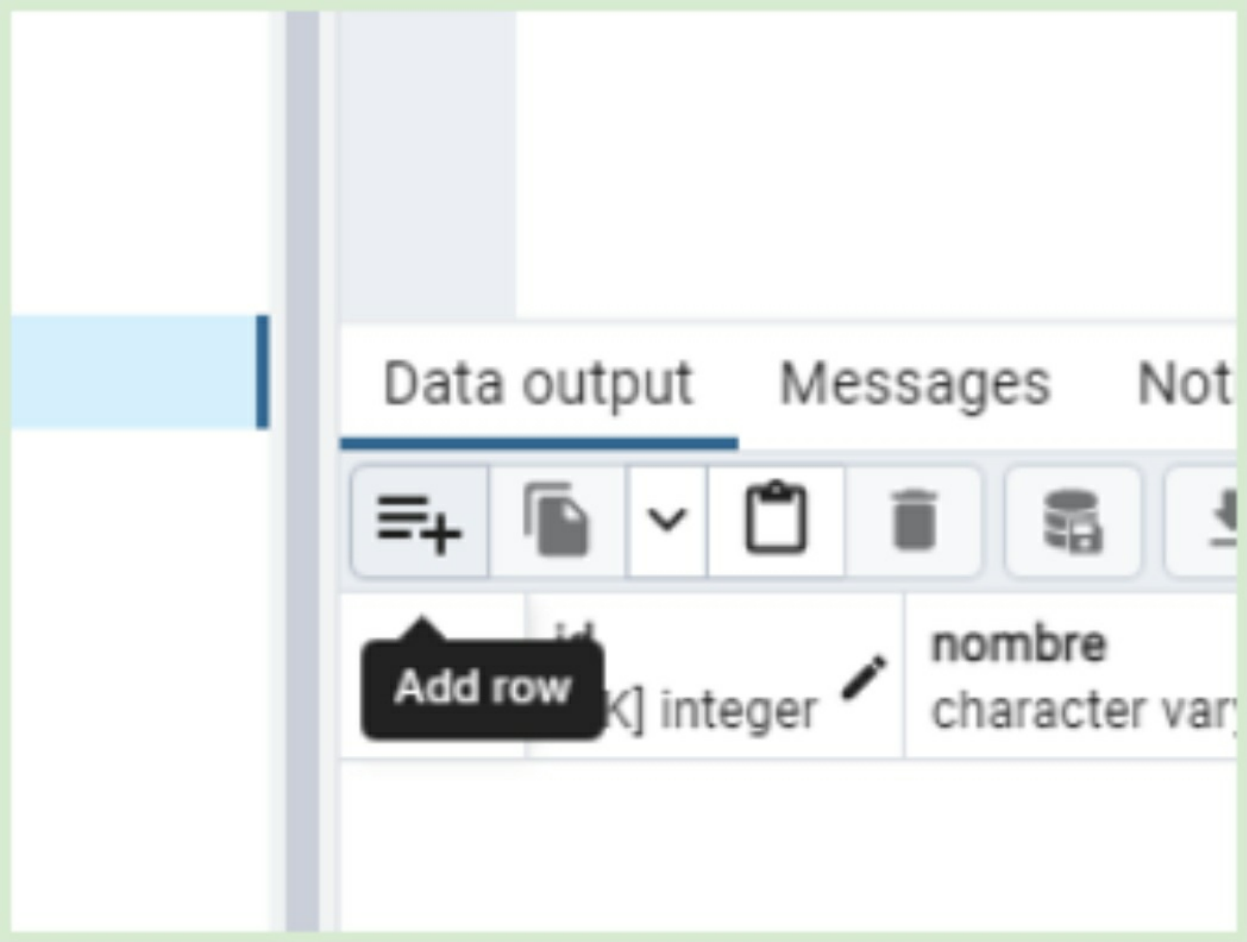
Ahora solo resta probar el nuevo servicio. Primero recordemos "setear" la clave *"productos.estrategia"* en *"EN_BD"* en el archivo *"application.properties"*.

```
productos.estrategia=EN_BD
```

Cargaremos algunas filas en nuestra tabla "producto" utilizando la herramienta PgAdmin.

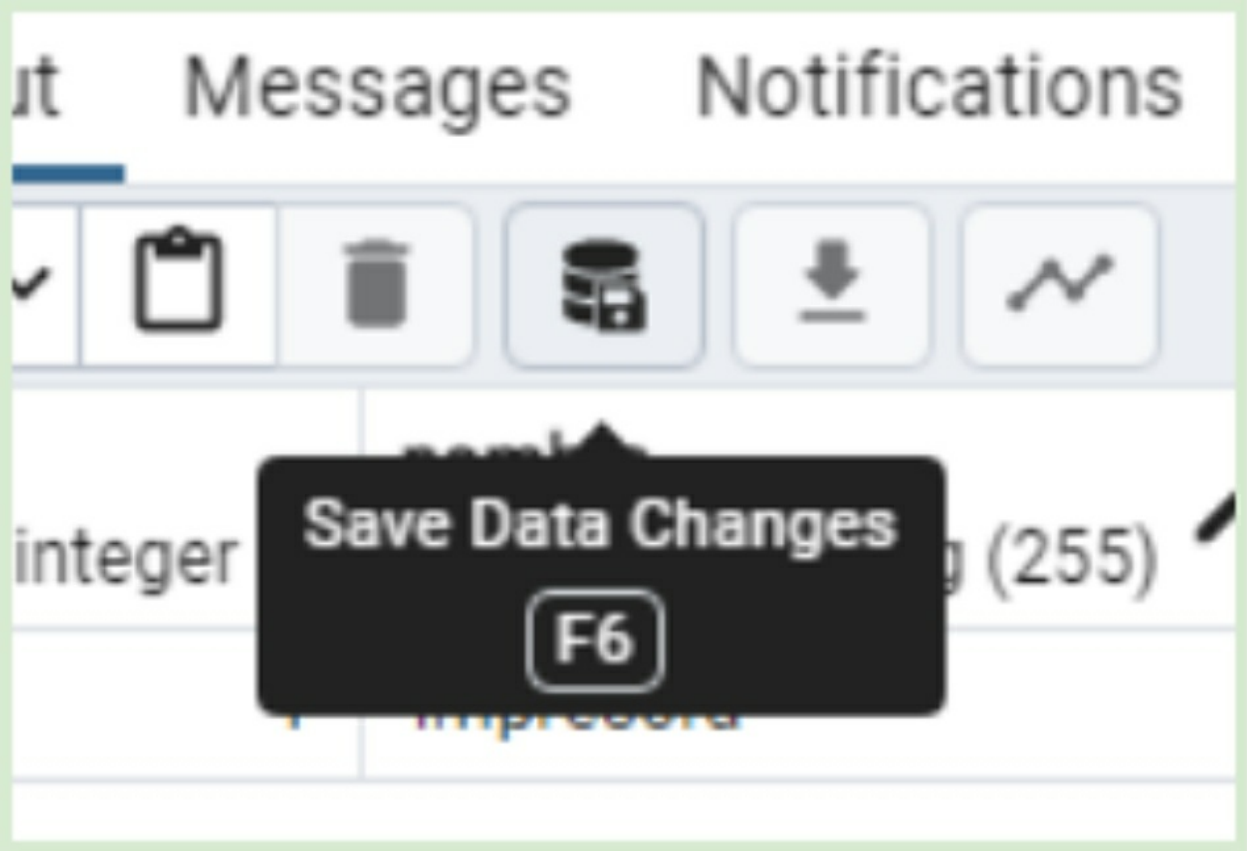


Situados en la opción “tables” y luego en la tabla “productos” de nuestra base “tienda”, hacemos click en el botón derecho del mouse, y en el menú flotante seleccionamos “view/edit Data” y luego “all rows”. Se nos cargará en el frame central una pantalla con varias opciones para manipular la tabla de productos.



En el frame inferior hacemos click en el primer icono para agregar una fila y cargamos los datos de un producto ficticio.

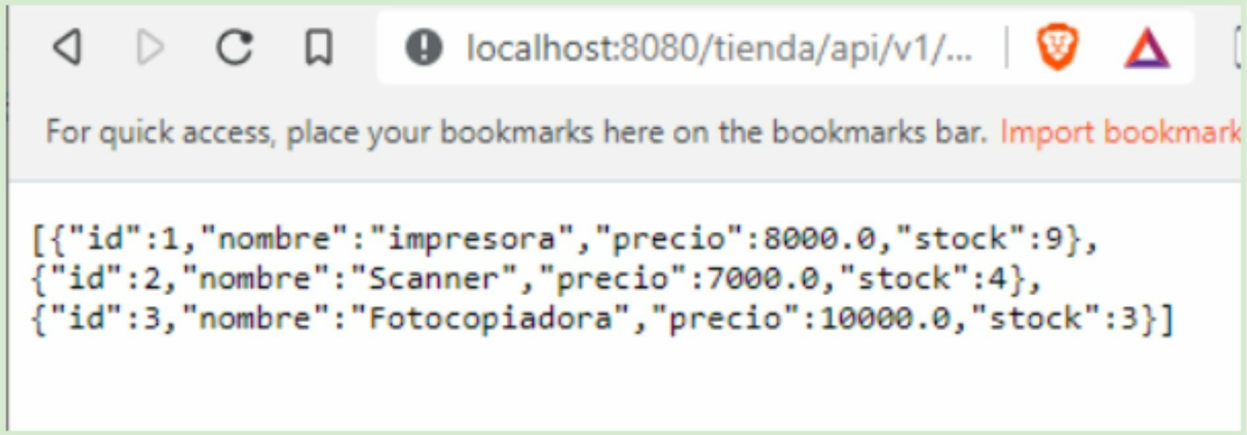
Y hacemos click en el icono que tiene una base de datos con un “diskette” para guardar los cambios.



Repetimos el procedimiento para guardar dos productos más tal que nos quede la tabla cargada de la siguiente manera.

	id [PK] integer	nombre character varying (255)	precio double precision	stock integer
1	3	Fotocopiadora	10000	3
2	2	Scanner	7000	4
3	1	impresora	8000	9

Ahora si reiniciamos Spring Boot y atacamos al endpoint que recupera todos los productos http://localhost:8080/tienda/api/v1/productos, deberíamos obtener este tipo de respuesta.



“Felicitaciones”

Si hemos logrado visualizar este resultado es que estamos cumpliendo un hito con Spring Boot. Hemos desarrollado una arquitectura completa de tres capas diferenciando objetos de modelo con entidades.

Ahora solamente presentaremos una forma más elegante orientado a streams y lambdas de gestionar los productos en nuestra clase servicio. Vale aclarar que esto es solo una alternativa y el lector es libre de optar por la fórmula que le sea más cómoda.

```
1 usage
public List<Producto> getProductos() {
    List<Producto> productos = productosRepository.findAll().
            stream().map(productoEntity -> {
                Producto producto = new Producto();
                producto.setId(productoEntity.getId());
                producto.setNombre(productoEntity.getNombre());
                producto.setPrecio(productoEntity.getPrecio());
                producto.setStock(productoEntity.getStock());
                return producto;
            }).collect(Collectors.toList());

    return productos;
}
```

5.3.5 Alta de producto

Vamos a implementar el alta producto. Para eso vamos a declarar en nuestra interfaz *"ProductService"* un nuevo método *"saveProducto(Producto producto)"*.

```java
package edu.tienda.core.services;

import edu.tienda.core.domain.Producto;

import java.util.List;

6 usages  4 implementations  4 related problems
public interface ProductoService {

    1 usage  4 implementations
    public List<Producto> getProductos();

    public void saveProducto(Producto producto);
}
```

Como de esta interfaz implementan ahora un total de cuatro clases concretas, se darán cuenta el porqué de los errores que reporta IntelliJ. Entonces, manos a la obra. Debemos dar implementación de este método a todas estas cuatro clases para que el proyecto pueda compilar.

Usaremos este método para las cuatro clases.

```java
@Override
public void saveProducto(Producto producto) {

}
```

Solo en la clase *"ProductoServiceDBImpl"* le daremos una implementación de la siguiente manera.

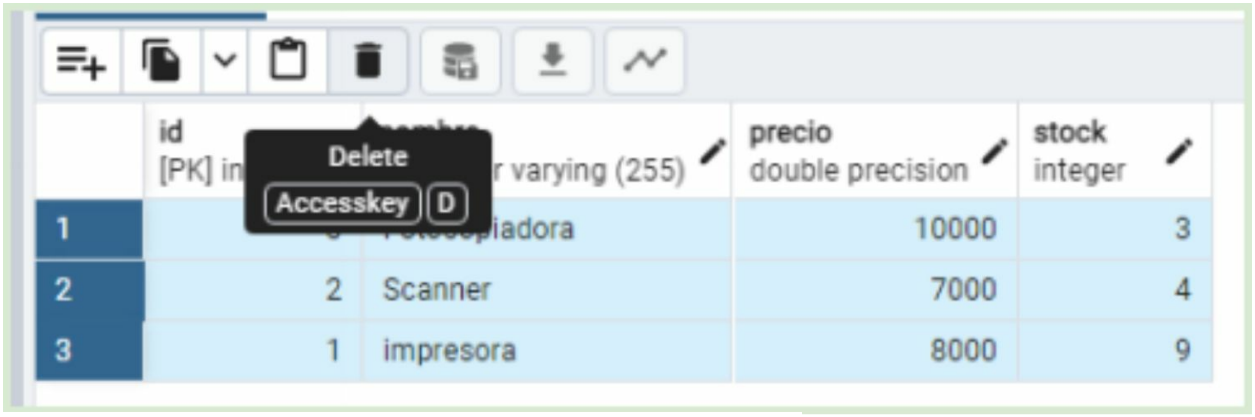
```java
@Override
public void saveProducto(Producto producto) {
    //Mapeo de Producto a ProductoEntity
    ProductoEntity productoEntity = new ProductoEntity();
    productoEntity.setNombre(producto.getNombre());
    productoEntity.setPrecio(producto.getPrecio());
    productoEntity.setStock(producto.getStock());

    //Persistencia
    productosRepository.save(productoEntity);
}
```

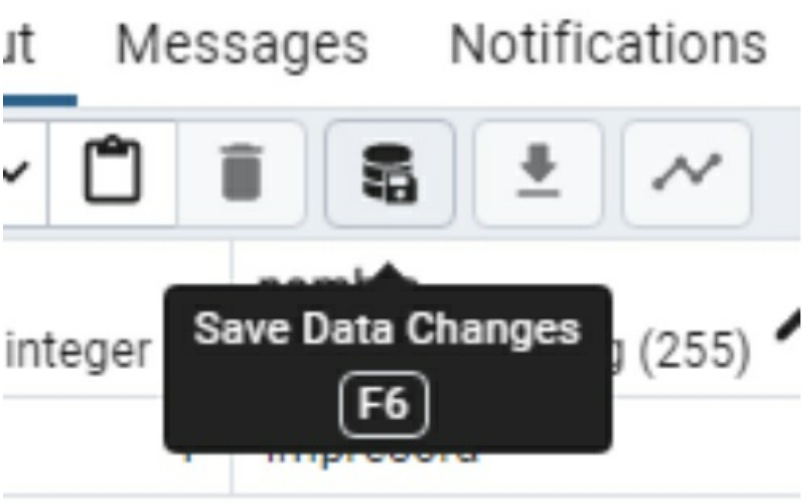
Observemos que primero debemos mapear manualmente el objeto de la clase *"Producto"* a un objeto de la clase *"ProductoEntity"* que es el que puede manejar la interfaz *"ProductRepository"*. En este caso hemos implementado el procedimiento inverso al método que recupera los productos.

Finalmente se presenta la simplicidad de JPA Repository para guardar el producto. En idioma de base de datos, lo que estamos haciendo es crear una nueva fila en la tabla "productos". Tenemos todo listo, en realidad casi listo porque todavía necesitamos implementar un endpoint en nuestro controlador de productos para manejar el alta. Lo haremos de forma similar a la implementación POST del alta de clientes.

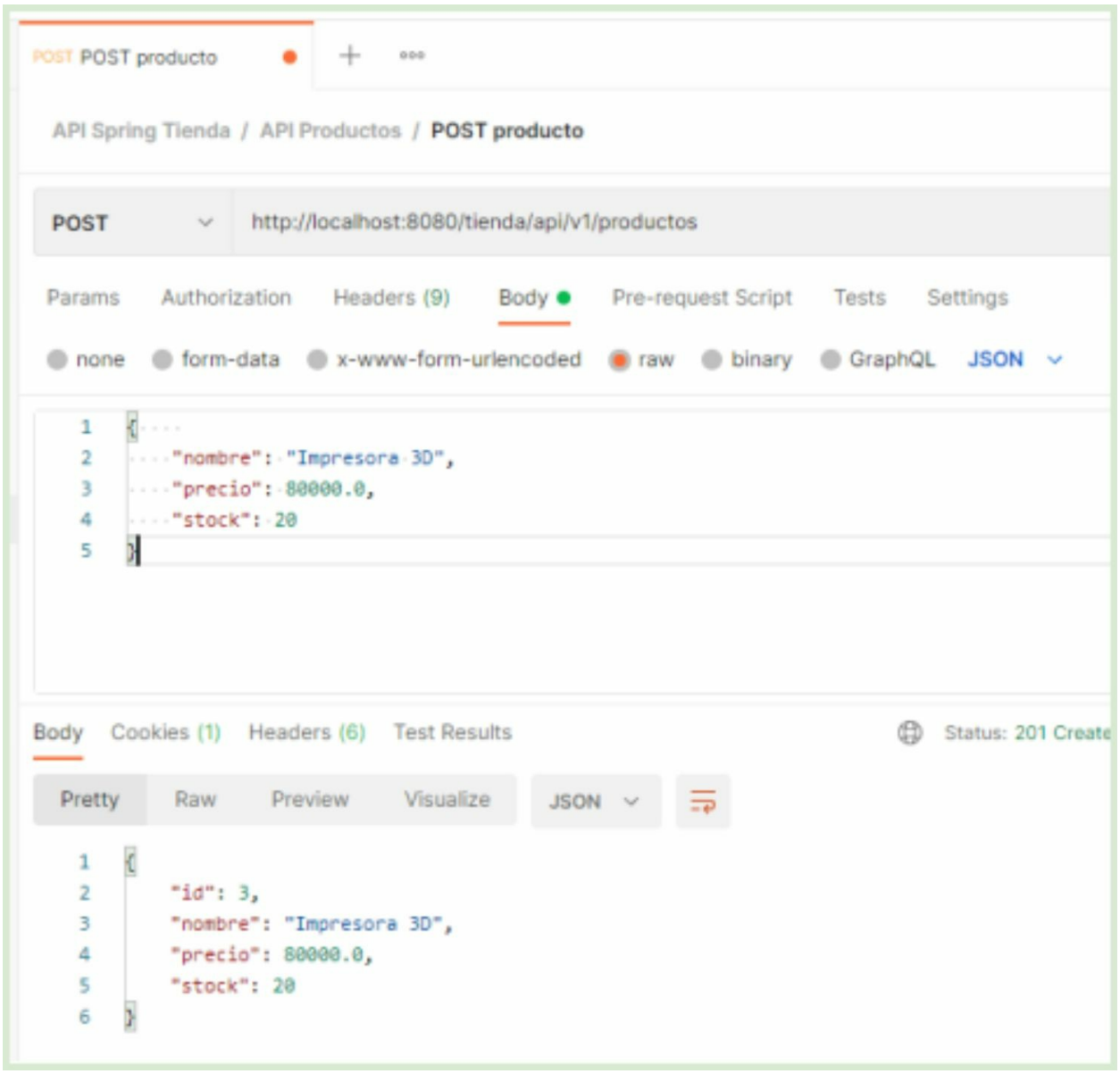
```java
@PostMapping
public ResponseEntity<?> altaProducto(@RequestBody Producto producto){

    productosService.saveProducto(producto);

    //Obteniendo URL de servicio.
    URI location = ServletUriComponentsBuilder
            .fromCurrentRequest()
            .path("/{id}")
            .buildAndExpand(producto.getId())
            .toUri();

    return ResponseEntity.created(location).body(producto);
}
```

Borramos todo el contenido de la tabla “productos” con PgAdmin seleccionando todas las filas y presionando el botón con icono de “cesto”. Luego presionamos el botón de confirmar datos.





Reiniciamos Spring y retomamos a nuestro fiel amigo Postman para la prueba de servicios de tipo POST. Creamos en Postman un “request” para atacar este servicio tal como se muestra a continuación.



Observar que no hemos suministrado el “id” ya que es JPA quien lo debería crear por nosotros utilizando la estrategia secuencial. Ejecutamos el “Request” de POSTMAN y observamos con el PgAdmin como se ha creado el producto en la tabla.

	id [PK] integer	nombre character varying (255)	precio double precision	stock integer
1	3	Impresora 3D	80000	20

Delegamos en el lector la implementación de la baja y modificación de los productos. La que se podrá implementar utilizando los métodos *“deleteById”* para eliminar y nuevamente *“save”* de la interfaz *“ProductoRepository”* para la modificación.

En el caso de la edición del producto, primero se deberá recuperar el mismo mediante el método “*findById”* disponible también en la interfaz JPA.

3.4 Capa Persistencia - JPA Wildcards

En esta sección implementaremos queries alternativos utilizando las JPA Wildcards.

5.4.1 JPA Wildcards

Si bien los repositorios JPA nos ofrecen los métodos básicos para manipular nuestras tablas con alta, baja, modificación, eliminación y recupero. En un sistema real, será necesario implementar algunas consultas que filtran filas por columnas.

Para esto existen los JPA wildcards, que con una simple definición de métodos abstractos en nuestras interfaces JPA nos permitirán fácilmente filtrar filas. Vamos a preparar primero la tabla. En principio crearemos algunos productos utilizando el mismo endpoint con postman que desarrollamos en la sección anterior para el alta.

Damos de alta varios productos, tal que nuestra tabla esté cargada de la siguiente manera.

	id [PK] integer	nombre character varying (255)	precio double precision	stock integer
1	3	Impresora 3D	80000	20
2	4	Monitor	9000	3
3	5	PC	50000	1
4	6	Mouse	100	1
5	7	Teclado	500	1

Ahora vamos a implementar un método en nuestra interfaz *"ProductoRepository"* para recuperar solamente aquellos productos con precio menor a 1000. Situados en la interfaz *"ProductoRepository"* escribimos la signatura del nuevo método de la siguiente manera.

```
2 usages
@Repository
public interface ProductosRepository extends JpaRepository<ProductoEntity,Integer> {

    1 usage
    List<ProductoEntity> findByPrecioLessThan(Double precio);

}
```

Observar con detenimiento el nombre del nuevo método. Porque el nombre lo "dice todo". En esa firma se expresa de forma clara que "query" debe generarse para recuperar los

productos. Exhibimos la signatura del método separándolo por colores de acuerdo a las palabras claves que contiene.

findByPrecioLessThan

La primera palabra clave es *"findBy"* que indica que necesitamos recuperar productos. No guardar, no eliminar sino recuperar. La segunda palabra indica *"Precio"* que está escrita exactamente como se define el atributo *"precio"* en nuestra entidad JPA de producto. Hasta aquí *"findByPrecio"* se traduce como "buscar productos por precio" o "buscar filas de productos por la columna precio".

"LessThan" ya es el filtro en sí y representa la cláusula que se aplicará para recuperar los productos por precio. La traducción literal es "menor que". Nuestra frase entera es entonces "Buscar productos por precio menor a …"

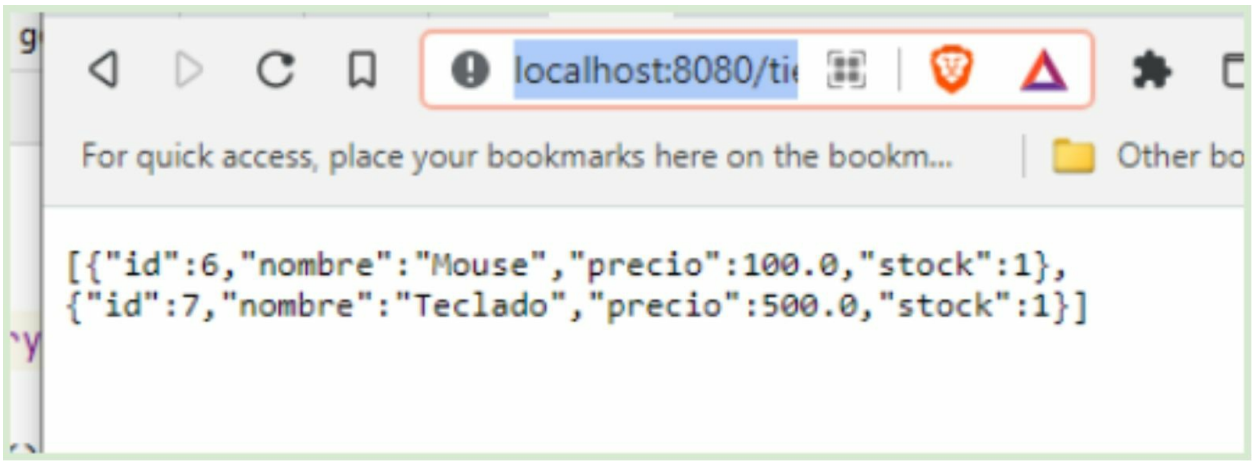
¿Menor a qué? Menor al valor que se ingresa por parámetro en este método, que es una variable de tipo *"Double"*. Igual a como está definido el atributo en nuestra entidad. Obviamente el retorno declarado de este método es una lista de productos ya que pueden haber más de una ocurrencia para este filtrado.

Hacemos un cambio en nuestra clase de servicio para recuperar esos productos pero ahora ejecutando el nuevo método con filtrado por precio de la siguiente manera.

```
public List<Producto> getProductos() {
    List<Producto> productos = productosRepository.findByPrecioLessThan(1000.0).
            stream().map(productoEntity -> {
                Producto producto = new Producto();
                producto.setId(productoEntity.getId());
                producto.setNombre(productoEntity.getNombre());
                producto.setPrecio(productoEntity.getPrecio());
                producto.setStock(productoEntity.getStock());
                return producto;
            }).collect(Collectors.toList());

    return productos;
}
```

Obviamente que si se trata de un sistema real deberíamos crear un nuevo endpoint Rest para este servicio y un nuevo método en *"ProductServiceDBImpl"* para este propósito. Pero a fin de agilizar las pruebas, en este caso haremos un "hook" editando solamente la llamada desde el servicio.

Ejecutamos nuevamente el endpoint para obtener los productos con precio menor a 1000: http://localhost:8080/tienda/api/v1/productos



¡Fantástico! No hemos tipeado ni un solo query SQL. Siempre nos hemos manejado en el paradigma orientado a objetos y en este caso, de manera declarativa ya que incluso no hemos tipeado ningún algoritmo complejo o requerido iterar sobre algún cursor como en viejas épocas.

5.4.2 Otros wildcards

Declaramos un nuevo método en nuestra interfaz *"ProductoRepository"* para obtener los productos por su nombre.

```
2 usages
@Repository
public interface ProductosRepository extends JpaRepository<ProductoEntity,Integer> {

    List<ProductoEntity> findByPrecioLessThan(Double precio);

    1 usage
    List<ProductoEntity> findByNombreLike(String nombre);

}
```

Verificar bien el nombre del nuevo método, *"findByNombreLike"* en este caso siempre siguiendo la nomenclatura de JPA wildcard, indica que se desea recuperar productos por nombre que sean parecidos a… el valor que llegará en el parámetro nombre. Destacar que en la firma del método figura *"nombre"* tal cual se llama el atributo en la clase *"ProductoEntity"*. Modificamos nuestro servicio para probarlo de la siguiente manera.

```
public List<Producto> getProductos() {
    List<Producto> productos = productosRepository.findByNombreLike("Teclado").
            stream().map(productoEntity -> {
                Producto producto = new Producto();
```

Reiniciamos Spring y ejecutamos nuevamente el endpoint
http://localhost:8080/tienda/api/v1/productos

```
[{"id":7,"nombre":"Teclado","precio":500.0,"stock":1}]
```

Nuestro resultado es solo un producto. El *"teclado"*. Finalmente probaremos otro wildcard más complejo que incluya dos cláusulas. Ahora filtraremos productos por precio y stock. Para este caso, vamos a consultar solamente aquellos cuyo precio sea mayor a 1000 y el stock sea menor a 10 unidades.

```
2 usages
@Repository
public interface ProductosRepository extends JpaRepository<ProductoEntity,Integer> {

    List<ProductoEntity> findByPrecioLessThan(Double precio);

    List<ProductoEntity> findByNombreLike(String nombre);

    1 usage
    List<ProductoEntity> findByPrecioGreaterThanAndStockLessThan(Double precio,Integer stock);

}
```

Observar la nueva firma del método. En este caso se filtra por dos columnas (“precio” y “stock”). En el primer segmento de la firma indica *“PrecioGreaterThan”* que se traduce como “precio mayor a..” y ese “mayor a” será establecido por el valor del primer parámetro que es el *“precio”*. Luego encontraremos una “wildcard” que se llama *“And”* y equivale sintácticamente al famoso *“AND”* utilizado en SQL. La firma sigue ahora indicando que se quiere filtrar por *“stock”* donde sea menor al valor del segundo atributo.

En total, *“findByPrecioGeaterThanAndStockLessThan”* se traduce como “buscar productos por precio menor a ‘precio’ y stock mayor a ‘stock’”. Modificamos el servicio para ejecutar este nuevo método asignando arbitrariamente un precio de 1000 y un stock de 10 unidades.

```
uctos() {
 = productosRepository.findByPrecioGreaterThanAndStockLessThan( precio: 1000.0, stock: 10).
uctoEntity -> {
ucto = new Producto();
Id(productoEntity.getId());
Nombre(productoEntity.getNombre());
```

Reiniciamos Spring y ejecutamos nuevamente el endpoint
http://localhost:8080/tienda/api/v1/productos


```
[{"id":4,"nombre":"Monitor","precio":9000.0,"stock":3},
{"id":5,"nombre":"PC","precio":50000.0,"stock":1}]
```


Recuperamos sólo aquellos productos con precio mayor a 10000 pero con stock menor a 10 unidades.

5.4.3 Queries nativos

Si bien las JPA wildcards disponibles son bastante prácticas y completas, es cierto que en nuestros sistemas reales vamos a toparnos con la necesidad de implementar consultas mucho más complejas en cuanto a la conjunción de tablas y cláusulas. En este caso, nos veremos un tanto limitados por las wildcards e incurrimos en la necesidad de declarar queries nativos SQL.

Spring Data está preparado para estos casos y dispone de una anotación @Query para dicho propósito como se puede ver en la siguiente figura.

```
@Query(value="select * from teamdto where name = ?1 and year >= ?2 and year <= ?3" ,nativeQuery = true)
public List<UserDTO> findAllTeamsBetweenYearAndName
(String name,int yearBegin,int yearEnd);
```

Delegamos en el lector la prueba de un método con query nativa para el recupero de productos.

5.4.4 Material descargable

Se podrá descargar todo el proyecto Spring Boot implementado en este libro desde la dirección web https://gitlab.com/veronikait/spring-boot-microservice

Spring Boot

Arquitectura de Back End

El desarrollo de microservicios con Spring Boot es una de las tecnologías más demandadas actualmente en la industria del software.

En este libro se desarrollará paso a paso un microservicio back end de Spring Boot completo utilizando IntelliJ

En esta edición se utilizan las últimas versiones tanto de Java, Spring y Spring Boot actualizadas a Septiembre de 2022. Se explicará con minucioso detalle cada una de las anotaciones que Spring nos provee para potenciar nuestro código fuente.

Pero sobre todas la cuestiones, se hará especial foco en el diseño y la arquitectura de cada capa de la solución. Entendiendo complejos conceptos como la inyección de dependencias, el bajo acoplamiento y la cohesión de los diversos componentes.

Cada implementación de controladores, servicios y persistencia se codifican empleando las mejores prácticas que nos brinda java.

Además integraremos en el microservicio tecnologías como Docker, PostgreSQL y Lombok. Realizaremos todas las pruebas de una API Rest bien formada con Postman.

Este libro incluye el proyecto entero descargable y listo para ejecutar.

Rafael Benedettelli