

Taller 3

Fecha: Octubre de 2024

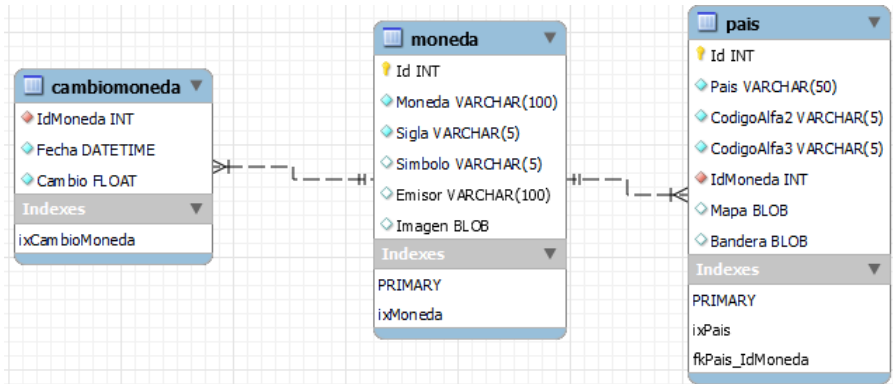
Indicador de logro a medir: Aplicar los conceptos de la lógica de programación, de la orientación a objetos, de la orientación a servicios y de la arquitectura del software en el desarrollo de una aplicación para Internet que acceda a una base de datos relacional y se integre con otros servicios.

NOTAS:

- Este taller se debe hacer como preparación para evaluaciones. En ningún caso representará una calificación.
- Los primeros ejercicios se entregan resueltos como ejemplo para el desarrollo de los demás

Elaborar la respectiva aplicación en Spring Boot y PostgreSQL para los siguientes enunciados:

1. El modelo relacional de una base de datos para registro de las monedas del mundo y sus respectivos cambios con respecto al dólar, es el siguiente ➔



Se desea una API que permita realizar las siguientes operaciones:

- Buscar la moneda de un país

API Monedas / Buscar Moneda por Pais

GET Send

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

Type: Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about [Bearer Token](#) authorization.

Token: eyJhbGciOiJIUzI1NiJ9.eyJzdWwiOiJmcmF5b3NvcmlvliwiaWF0IjoxNzEzMDQwMTExLCJleHAiOiE3MTMwNDU5MTF9.i-oTszEfhUx-keAQeTCjC4n4DJQukkKKI50QaojUzQQ

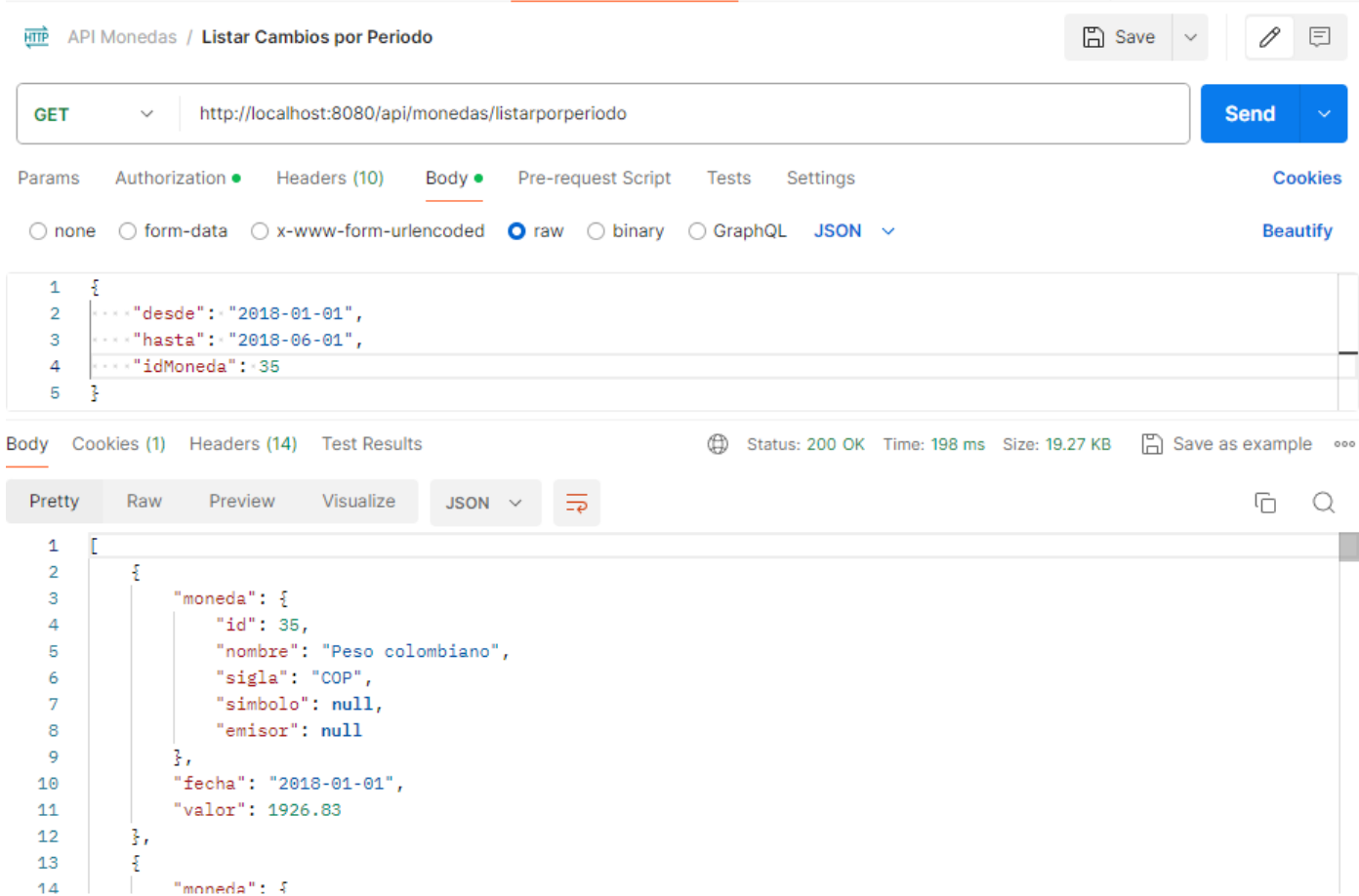
Body Cookies (1) Headers (14) Test Results Status: 200 OK Time: 199 ms Size: 502 B Save as example

Pretty Raw Preview Visualize JSON

```

1 {
2   "id": 35,
3   "nombre": "Peso colombiano",
4   "sigla": "COP",
5   "simbolo": null,
6   "emisor": null
7 }
```

- Listar los cambios que ha tenido una moneda en un rango de fechas



The screenshot shows a Postman interface for a REST client. The request is a GET to `http://localhost:8080/api/monedas/listarporperiodo`. The body is a JSON object with the following structure:

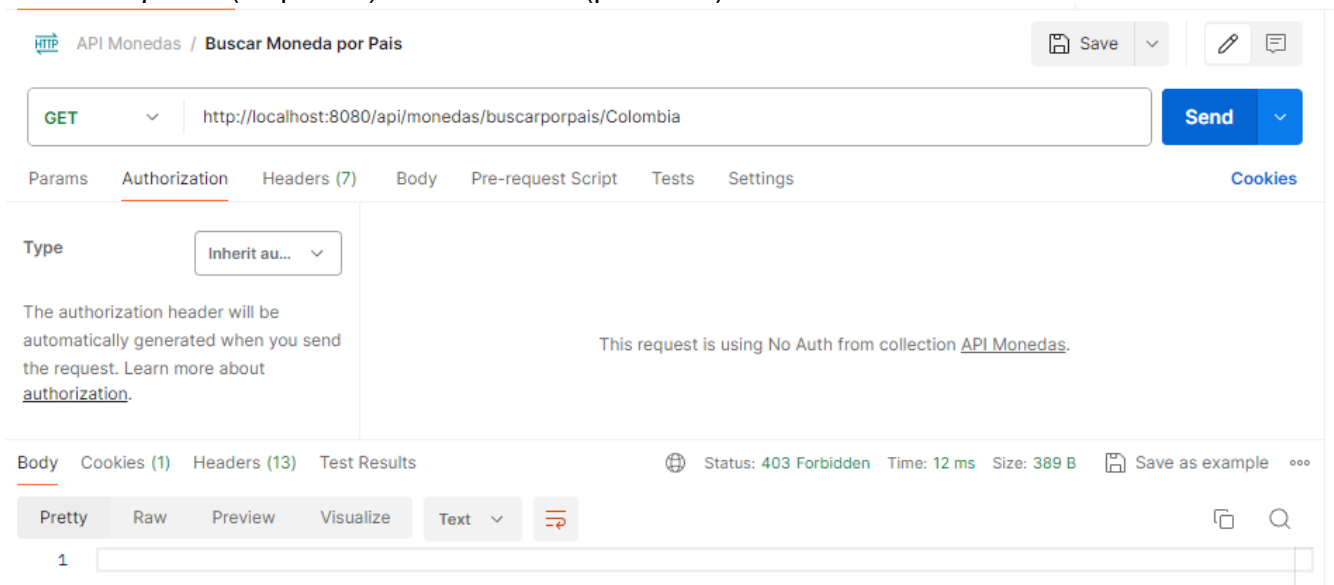
```
1 {
2   "desde": "2018-01-01",
3   "hasta": "2018-06-01",
4   "idMoneda": 35
5 }
```

The response is displayed in the 'Body' tab, showing a JSON array with one object:

```
1 [
2   {
3     "moneda": {
4       "id": 35,
5       "nombre": "Peso colombiano",
6       "sigla": "COP",
7       "simbolo": null,
8       "emisor": null
9     },
10    "fecha": "2018-01-01",
11    "valor": 1926.83
12  },
13  {
14    "moneda": {
```

Metadata: Status: 200 OK, Time: 198 ms, Size: 19.27 KB.

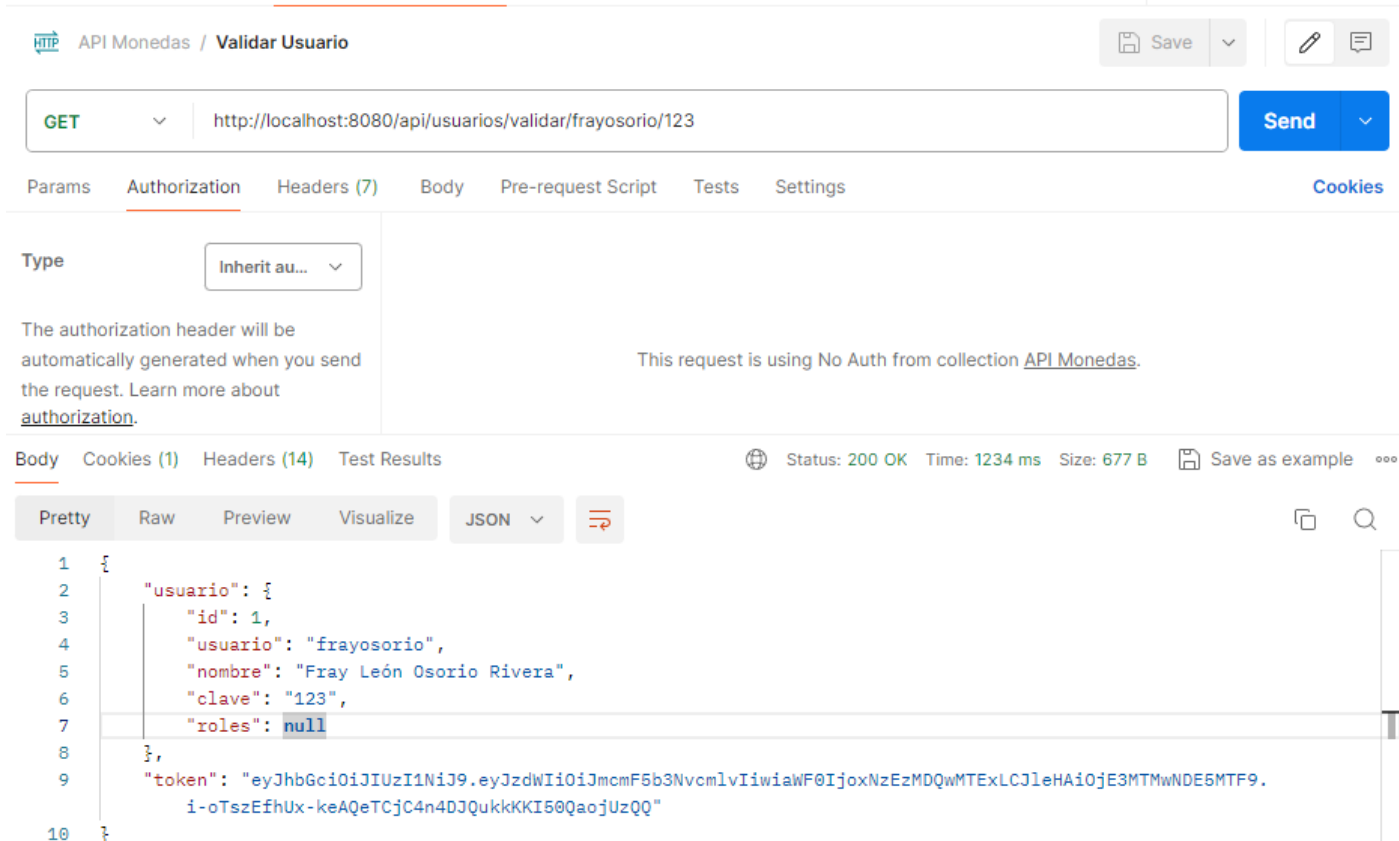
Como se puede observar en los *request* (solicitudes), se debe proporcionar en ellas el respectivo token de seguridad, que, en caso de no hacerse, se obtendrá un *response* (respuesta) **403 Forbidden** (prohibido):



The screenshot shows a Postman interface for a REST client. The request is a GET to `http://localhost:8080/api/monedas/buscarporpais/Colombia`. The response is displayed in the 'Body' tab, showing a 403 Forbidden status.

Metadata: Status: 403 Forbidden, Time: 12 ms, Size: 389 B.

- Por ello, se debe realizar también un método web que permita validar las credenciales de usuario (*login*) y devuelva tanto el objeto del usuario correspondiente como el token de seguridad aceptado:



API Monedas / Validar Usuario

GET http://localhost:8080/api/usuarios/validar/frayosorio/123

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies

Type Inherit au... The authorization header will be automatically generated when you send the request. Learn more about [authorization](#).

This request is using No Auth from collection [API Monedas](#).

Body Cookies (1) Headers (14) Test Results Status: 200 OK Time: 1234 ms Size: 677 B Save as example

Pretty Raw Preview Visualize JSON

```
1 {
2   "usuario": {
3     "id": 1,
4     "usuario": "frayosorio",
5     "nombre": "Fray León Osorio Rivera",
6     "clave": "123",
7     "roles": null
8   },
9   "token": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJmcmF5b3NvcmIvIiwiaWF0IjoxNzEzMDQwMTExLCJleHAiOiE3MTMwNDU1MTF9.
10  i-oTszEfhUx-keAQeTCjC4n4DJQukkKKI50QaojUzQQ"
```

R/

Para desarrollar este aplicativo web se tendrán las siguientes definiciones:

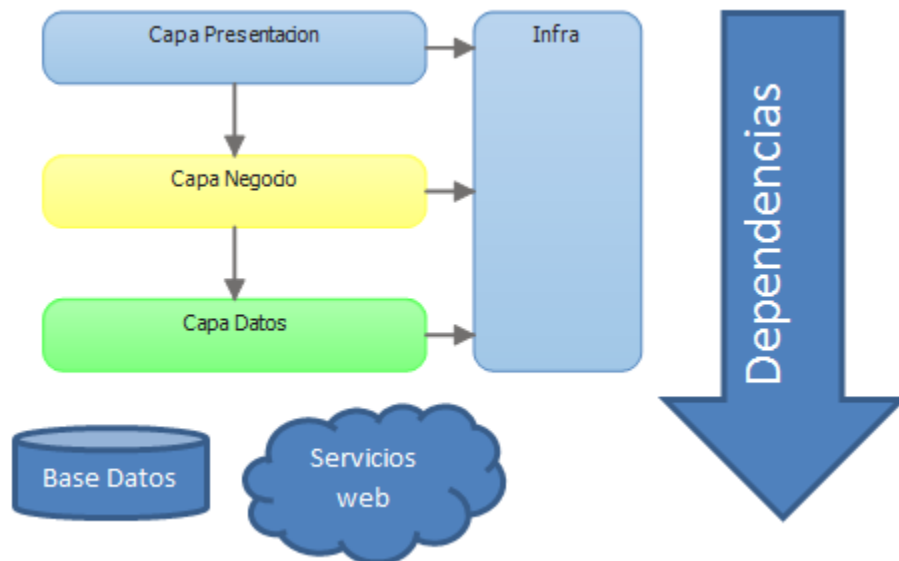
¿Qué es la Arquitectura Onion?

Introducción

El termino **Onion Architecture** o **Arquitectura cebolla** fue acuñada por Jeffrey Palermo, siendo su primer artículo publicado en el año 2008 donde habla de este tema. En este artículo al autor señala que este tipo de arquitectura es ideal para aplicaciones con un comportamiento complejo del negocio o que van a tener un tiempo de vida largo. Se debe tener en cuenta que este tipo de arquitectura no es ideal para aplicaciones pequeñas, si se aplica a una aplicación que solo realiza operaciones CRUD o no tiene nada de complejidad obtendremos como resultado que nuestro diseño sufra del *design smell* de *Complejidad innecesaria*. La *Onion Architecture* nació por el gran problema que existe con el muy usado estilo **NCapas** en donde las capas superiores dependen de las inferiores y de algunos temas de infraestructura (base de datos, servicios web, servidores de correo, servidores ftp, etc), pero el principal punto en contra que tiene el estilo *NCapas* es el acoplamiento innecesario que genera entre las capas (inferiores y superiores) y entre varias responsabilidades de infraestructura. Además, se debe mencionar que la *Onion*

Architecture está muy relacionado con el principio **SOLID *Dependency Inversion*** por lo que es necesario contar con algún mecanismo de **inyección de dependencias**.

A continuación, veamos las características del estilo tradicional *NCapas* contra *Onion Architecture*.

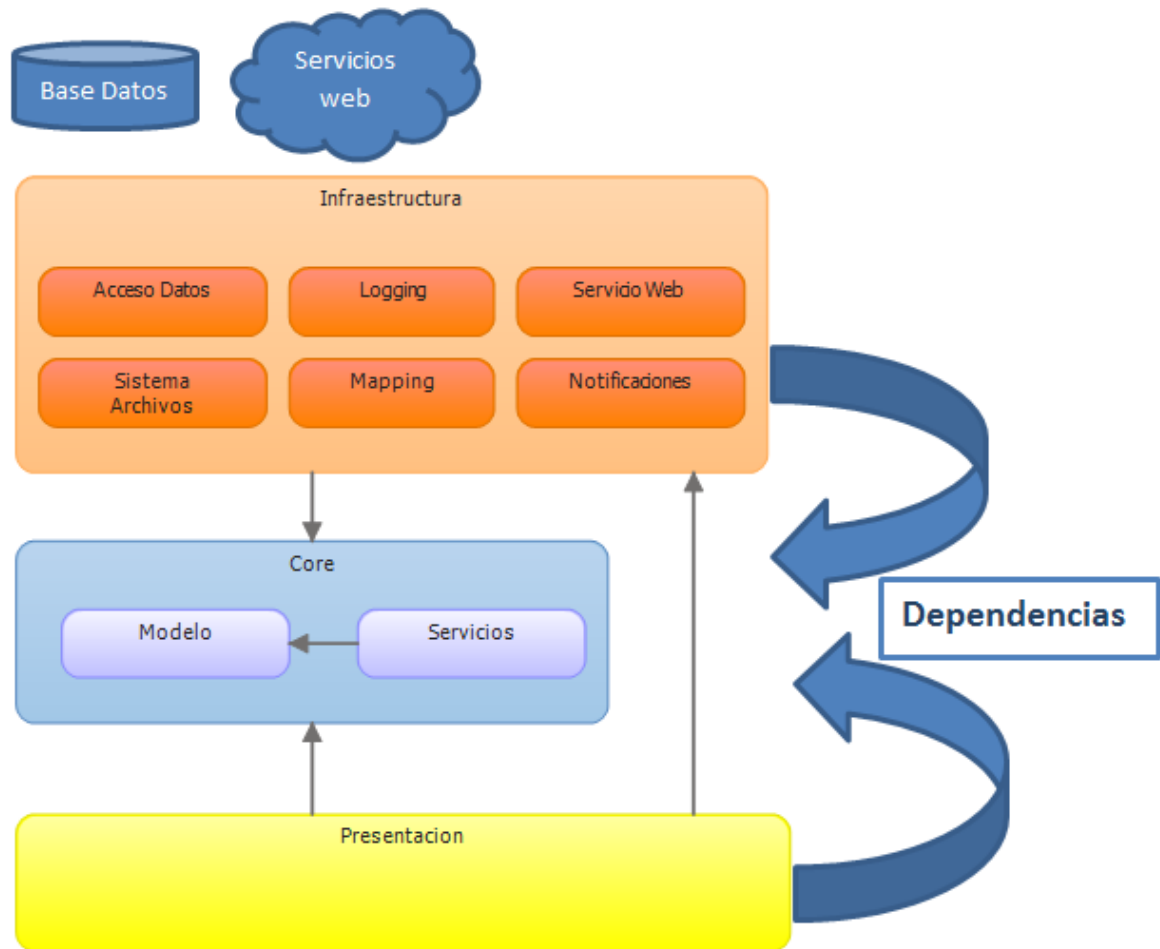


Problemas

- Cada capa está acoplada a la capa inferior, este diseño crea un acoplamiento innecesario.
- Cada capa depende de algunos temas de infraestructura.
- La capa de presentación depende indirectamente de la capa de datos, cualquier cambio que ocurra en base de datos tendrá un impacto sobre las capas superiores.
- La capa de presentación no puede funcionar si la capa de negocio no está presente y la capa de negocio no funciona si no existe la capa de datos.
- Este diseño viola el principio **SOLID *Dependency Inversion*** que indica lo siguiente: “*Abstractions should not depend upon details. Details should depend upon abstractions*”. En este sentido podemos afirmar que la capa de negocio (abstracción) no debe depender de la de datos (detalle). Los detalles deben depender de las abstracciones.

Onion Architecture

La regla principal es que todo el código debe depender de las capas centrales y no de las externas.



Finalidad

La finalidad de este estilo de arquitectura es poder construir aplicaciones que sean fáciles de mantener, probar y sobre todo que se encuentren desacopladas de elementos de infraestructura tales como base de datos o servicios.

- **Independencia de frameworks:** La arquitectura no debe depender de ninguna librería. Esto permite usar frameworks como herramientas.
- **Testeable:** Las reglas de negocio pueden ser testeadas sin necesidad de conocer la interfaz de usuario, la base de datos o algún servicio externo.
- **Independiente de la UI:** La interfaz de usuario puede cambiar fácilmente sin tener que afectar al resto del sistema. Un tipo de interfaz de usuario puede ser reemplazada por otra sin cambiar las reglas de negocio.
- **Independiente de la base de datos:** Se puede cambiar de motor de base de datos sin ningún problema. Las reglas de negocio no deben estar amarradas a esta.
- **Independiente de agentes externos:** Las reglas de negocio no deben saber nada que este fuera de su contexto.

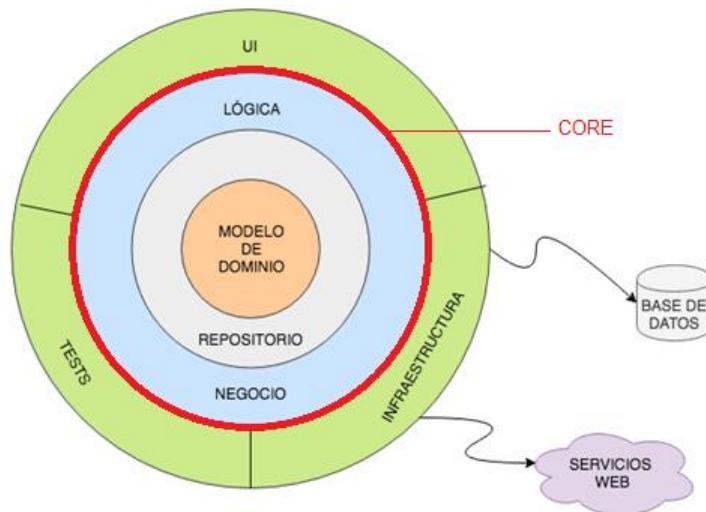
Características

- La comunicación entre las capas se realiza usando interfaces. Cualquier implementación es provista en tiempo de ejecución.
- Cualquier dependencia externa, como base de datos o servicios web, pertenecen a la capa externa.
- La UI también pertenece a la capa externa.
- Los objetos que representan el negocio pertenecen a las capas internas.
- Las capas externas dependen de las capas internas.
- Las capas internas no pueden depender de las capas externas.
- Todo lo que pueda cambiar se debe encontrar en una capa externa.

Elementos

El número de elementos puede variar, pero siempre se debe tener en cuenta que el **Domain Model** es el centro de todo.

- **Core:** *Domain model*, servicios, interfaces de repositorio, otras interfaces.
- **UI:** Mvc, Servicios Web, etc.
- **Test:** Pruebas unitarias o de integración.
- **Infraestructura:** Implementación de las interfaces definidas en el Core.

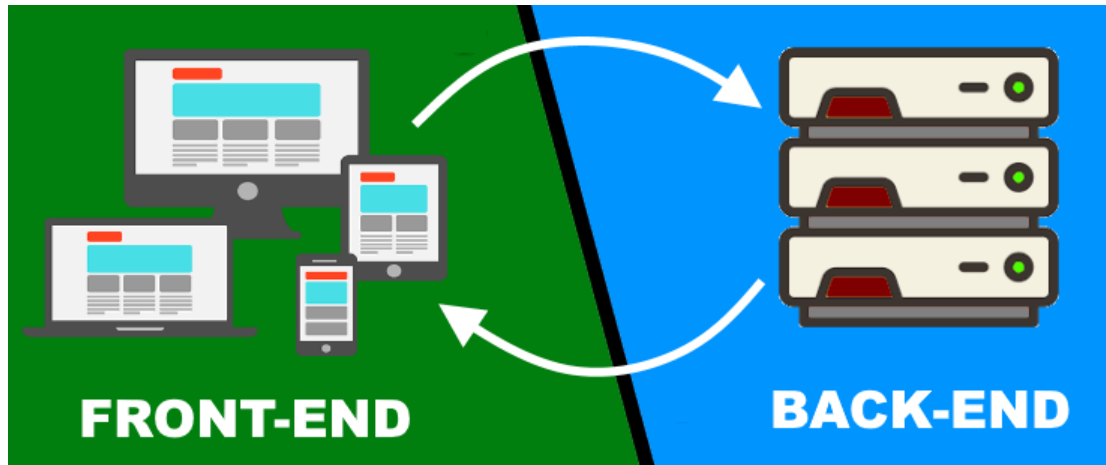


¿Qué es *Backend*?

Antes de empezar a hablar de *Backend*, es importante comprender el concepto de *Frontend* primero, ya que, siendo ambos conceptos indivisibles, el segundo nos será mucho más cercano y nos ayudará a comprender el primero más fácilmente.

En pocas palabras, el **Frontend** de una app es la parte que el usuario ve y toca. En él, se incluyen imágenes, botones, menús, transiciones, etc. Es la capa más superficial de la app, cuyos elementos son incapaces de funcionar por sí mismos. Es aquí donde el Backend interviene, dando vida a todo lo anterior.

El **Backend**, también conocido como *CMS* o *Backoffice*, es la parte de la app que el usuario final no puede ver. Su función es acceder a la información que se solicita, a través de la app, para luego combinarla y devolverla al usuario final.



¿Cómo funciona un *Backend*?

Las funciones del *Backend* son las siguientes:

- **Acceder a la información que se pide, a través de la app:** cuando se usa una aplicación web o móvil se pide información de manera continua, no importa si la app es de búsqueda de información, un juego o una red social. Esto implica que una parte de la app (el *Backend*), tiene que ser capaz de encontrar y acceder a la información que se solicite. El proceso de búsqueda de datos no es fácil, ya que estos se almacenan en grandes bases de datos, que se encuentran, además, protegidos para no exponer lo que en nuestra área se denomina información sensible. En este punto, un *Backend* bien diseñado debe ser capaz no sólo de encontrar la información precisa que el usuario requiere, sino también de acceder a ella de manera segura
- **Combinar la información encontrada y transformarla:** Una vez encontrada, el Backend combina la información para que resulte útil al usuario. Pongamos, como ejemplo, una aplicación de transporte y una orden de búsqueda: “cómo llegar del trabajo a casa”.

En este caso, la aplicación necesitará acceder a las bases de datos no sólo de todas las compañías de autobuses de la ciudad, sino también las de las empresas de taxis, metro y, por supuesto, Google Maps. La ingente cantidad de información con la que el *Backend* trabaja, hace que su diseño deba ser sumamente preciso, ya que debe ser capaz de encontrar y filtrar lo que es relevante de lo que no, para luego combinarlo de manera útil.

- **Devolver la información al usuario:** Finalmente, el *Backend* envía la información relevada de vuelta al usuario. Pero, ¿cuántos usuarios son capaces de leer datos escritos en código puro? Pocos. Es por ello que el *Backend* necesita de traductores capaces de convertir los datos escritos en código a lenguaje humano. Es aquí donde intervienen las famosas *APIs*, trabajando en conjunto con el *Frontend*.

En pocas palabras, las APIs son las herramientas encargadas de transportar la información desde el *Backend* hasta el *Frontend*, que es donde el proceso final de traducción toma forma, y donde la información escrita en código se convierte en los diseños, las imágenes, las letras y los botones que el usuario final entiende y con los que puede interactuar.

Este proceso es hecho por el *Frontend* en dos fases, que tienen lugar en dos subcapas que conforman su estructura:

- **La subcapa lógica**, relacionada con el lenguaje específico, en el que la app ha sido desarrollada. Como ya sabemos, las apps pueden ser desarrolladas para distintos sistemas operativos - iOS, Android, WindowsPhone...- que requieren ser escritas en distintos lenguajes de código. En este punto, la subcapa lógica del *Frontend* se encarga de hacer una primera traducción del lenguaje en el que las APIs envían la información al lenguaje específico de cada sistema operativo. Este es un proceso que, a pesar de ocurrir en el Frontend de nuestra app, el usuario final no ve. Podría llamarse el “Backend del Frontend”
- **La subcapa visual**, relacionada con el diseño estético de la app, en la que se encuentran todos los elementos que el usuario final puede ver. La capa visual del *Frontend* es donde, finalmente, se produce la conversión de la información a los elementos con el que el usuario final se relaciona

Para concluir con las funciones y procesos del *Backend*, el desarrollo de un *Backend* sólido es la clave para conseguir una buena experiencia de usuario. Se pueden desarrollar los mejores y más novedosos diseños, se puede tener la mejor idea para un negocio mobile pero, al final, si los cimientos de la app fallan, la aplicación acabará siendo un fracaso

Con base en lo anterior, un buen *Backend* de cumplir con las siguientes características:

- **Escalabilidad:** se refiere a la flexibilidad del mismo para integrarse a nuevas estructuras y códigos. Este es un aspecto esencial, especialmente si la app ha sido diseñada para utilizarse a largo plazo. ¿Por qué? Porque los cambios en la web y dispositivos móviles son inevitables. Cada día nuevos sistemas operativos y dispositivos móviles son lanzados al mercado, por no hablar de que el modelo de negocio puede - y debe – evolucionar
- **Seguridad:** Debido a la constante interacción del *Backend* con las bases de datos, desarrollar el código siguiendo prácticas seguras es sumamente importante, más aún si la app está diseñada para manejar información sensible, como datos personales, financieros o médicos. Para el desarrollo de un *Backend* seguro, se recomienda trabajar sólo con desarrolladores cualificados, hacer uso de conexiones seguras -como las famosas HTTPS- y utilizar bases de datos encriptadas
- **Robustez:** Se refiere a la "fuerza" del Backend, o su capacidad para funcionar en cualquier contexto. Imaginen, por ejemplo, que se desarrolla una app para ser un *mobile commerce*. En este caso, la aplicación deberá desarrollarse teniendo en cuenta situaciones inesperadas, como el registro y uso masivo de

la app en fechas de venta clave - Navidad, Rebajas -. Un *Backend* desarrollado de manera robusta asegura que, ante este tipo de situaciones, la app siga funcionando, evitando los famosos crashes, que dan lugar a malas experiencias de uso y por lo tanto a la temida desinstalación de la app.

¿Qué es una **API REST**?

Las **API** son mecanismos que permiten a dos componentes de software comunicarse entre sí mediante un conjunto de definiciones y protocolos. Por ejemplo, el sistema de software del instituto de meteorología contiene datos meteorológicos diarios. La aplicación meteorológica de su teléfono “habla” con este sistema a través de las API y le muestra las actualizaciones meteorológicas diarias en su teléfono.

Las **API REST** son las más populares y flexibles que se encuentran en la web actualmente. El cliente envía las solicitudes al servidor como datos. El servidor utiliza esta entrada del cliente para iniciar funciones internas y devuelve los datos de salida al cliente.

¿Qué es REST?

La **Transferencia de Estado Representacional** (en inglés **Representational State Transfer - REST**) es una arquitectura que se apoya en el estándar HTTP la cual permite crear aplicaciones y servicios que pueden ser usadas por cualquier dispositivo o cliente que utilice HTTP. Permite describir cualquier interfaz para obtener datos o indicar la ejecución de operaciones sobre los datos, en cualquier formato (XML, JSON, etc.)

Las operaciones más importantes en cualquier sistema **REST** (CRUD) son:

- GET (Leer y consultar los registros)
- POST (Crear nuevos registros)
- PUT (Editar y modificar los registros)
- DELETE (Eliminar los registros)

Las ventajas de usar protocolo **REST** es separar totalmente la interfaz de usuario del servidor y del almacenamiento de datos, esto contribuye a una mejora en la portabilidad de la plataforma y un aumento en la escalabilidad, así como aumentar la seguridad del mismo.

La principal característica de la **API REST** es la ausencia de estado. La ausencia de estado significa que los servidores no guardan los datos del cliente entre las solicitudes. Las solicitudes de los clientes al servidor son similares a las URL que se escriben en el navegador para visitar un sitio web. La respuesta del servidor son datos simples, sin la típica representación gráfica de una página web.

Una **API web** o **API de servicios web** es una interfaz de procesamiento de aplicaciones entre un servidor web y un navegador web. Todos los servicios web son API, pero no todas las API son servicios web. La **API REST** es un tipo especial de API web que utiliza el estilo arquitectónico estándar explicado anteriormente.

Los diferentes términos relacionados con las API, como API de Java o API de servicios, existen porque históricamente las API se crearon antes que la World Wide

Web. Las API web modernas son *API REST* y los términos pueden utilizarse indistintamente.

¿Cómo proteger una *API REST*?

Todas las API deben protegerse mediante una autenticación y una supervisión adecuadas. Las dos maneras principales de proteger las API de REST son las siguientes:

- **Tokens de autenticación:** Se utilizan para autorizar a los usuarios a hacer la llamada a la API. Los tokens de autenticación comprueban que los usuarios son quienes dicen ser y que tienen los derechos de acceso para esa llamada concreta a la API. Por ejemplo, cuando inicia sesión en el servidor de correo electrónico, el cliente de correo electrónico utiliza tokens de autenticación para un acceso seguro
- **Claves de API:** Verifican el programa o la aplicación que hace la llamada a la API. Identifican la aplicación y se aseguran de que tiene los derechos de acceso necesarios para hacer la llamada a la API en cuestión. Las claves de API no son tan seguras como los tokens, pero permiten supervisar la API para recopilar datos sobre su uso. Es posible que haya notado una larga cadena de caracteres y números en la URL de su navegador cuando visita diferentes sitios web. Esta cadena es una clave de la API que el sitio web utiliza para hacer llamadas internas a la API.

¿Qué es JSON?

La **Notación de Objetos de JavaScript** (en inglés **JavaScript Object Notation - JSON**) es un formato de texto sencillo para el intercambio de datos. Es básicamente un subconjunto de la notación de objetos de JavaScript, que se ha adoptado ampliamente como alternativa a XML llegando a ser un formato totalmente independiente del lenguaje.

Entre sus ventajas sobre XML como formato para intercambio de datos, está que resulta más simple escribir un analizador sintáctico (parser) para él. En JavaScript, un texto JSON se puede analizar fácilmente usando la función `eval()`, algo que por la universalidad del lenguaje (funciona en todos los navegadores) ha sido fundamental para haya sido aceptado por casi toda la comunidad de desarrolladores

La siguiente es la lista de tipos de datos disponibles en JSON:

Tipo	Descripción	Ejemplo
Números	Se permiten números negativos y opcionalmente pueden contener parte fraccional separada por punto	12 -128 789.12345
Cadenas de Texto	Conformado por secuencias de cero o más caracteres. Van entre comillas dobles y permiten incluir secuencias de escape	"Gabriel García Marquez"
Booleanos	Pueden tener dos valores: true o false	
Nulos	Representan el valor nulo null	
Vectores	Representa una lista de cero o más valores los cuales pueden ser de cualquier tipo. Los valores se separan por comas y el vector se mete entre corchetes.	["verde", "amarillo", "rojo"]
Objetos	Son colecciones de pares de la forma <nombre>:<valor> separados por comas y puestas entre llaves. El nombre tiene que ser una cadena de texto y el valor puede ser de cualquier tipo	{"ciudad": "Medellín", "departamento": "Antioquia", "país": "Colombia"}

¿Qué es MVC?

MVC es una propuesta de diseño de software utilizada para implementar aplicaciones donde se requiere el uso de interfaces de usuario. Surge de la necesidad de crear software más robusto con un ciclo de vida más adecuado, donde se potencie la facilidad de mantenimiento, reutilización del código y la separación de conceptos.

Su fundamento es la separación del código en tres capas diferentes, acotadas por su responsabilidad, en lo que se llaman **Modelos, Vistas y Controladores**, (en inglés **Model, Views & Controllers – MVC**). MVC es un concepto que ya tiene varias décadas y fue presentado incluso antes de la aparición de la Web, pero es en los últimos años que ha ganado mucha fuerza y seguidores gracias a la aparición de numerosos frameworks de desarrollo web que utilizan el patrón MVC como modelo para la arquitectura de las aplicaciones web.

Capa	Descripción
Modelos	Es la capa donde se trabaja con los datos, por tanto, contendrá las operaciones para acceder a la información y también para actualizar su estado. Los datos generalmente están en una base de datos, por lo que en los modelos tendremos todas las funciones que accederán a las tablas y harán los correspondientes <i>selects, updates, inserts</i> , etc. También se ha vuelto común que en lugar de usar directamente instrucciones SQL, que suelen depender del motor de base de datos con el que se esté trabajando, se utiliza un dialecto de acceso a datos basado en clases y objetos.
Vistas	Contiene el código de la aplicación que va a producir la visualización de las interfaces de usuario, o sea, el código que permitirá renderizar el despliegue en lenguajes como HTML. En la vista se trabaja con los datos provenientes de los modelos y con ellos se generará la salida, tal como la aplicación lo requiera.
Controladores	Contiene el código que responde a las acciones que se solicitan en la aplicación, como visualizar un elemento, registrar información, realizar una búsqueda de información, etc. En realidad, es una capa que sirve de enlace entre las vistas y los modelos, respondiendo a los mecanismos que puedan requerirse para implementar las necesidades de la aplicación. Sin embargo, su responsabilidad no es manipular directamente datos, ni mostrar ningún tipo de salida, sino servir de enlace entre los modelos y las vistas para implementar las diversas necesidades del desarrollo.

¿Qué es Spring Boot?

Java Spring Boot es una de las herramientas principales del ecosistema de desarrollo web Backend con Java, muy útil en las aplicaciones con características *Enterprise*, principalmente en arquitecturas basadas en servicios web (REST y SOAP) y microservicios



Antes que nada, hay que hacer énfasis en que *Spring Boot* NO es Spring y que este proyecto surge de la necesidad de hacer aplicaciones Java sin tantas complicaciones de configuración y toda la problemática que eso conlleva.

Por ello y por muchas razones más, nace *Spring Boot*, que junto a proyectos como *Spring framework*, *Spring Data*, *Spring Security*, *Spring Cloud*, entre otros, hacen la combinación perfecta para desarrollar, probar y desplegar nuestras aplicaciones en un entorno rápido, eficaz y bastante simple.

Las siguientes son las características y usos de *Spring Boot*:

- **Facilidad de despliegue con los servidores embebidos:** Con *Spring Boot* nos olvidamos de tener que desplegar artefactos *Jar* o *War* de manera independiente en uno o muchos servidores web diferentes. Porque nos provee una serie de *contenedores web servlet* para que se despliegue nuestra aplicación automáticamente solo con un “Run”
Así mismo, de estos contenedores web se puede elegir el que más convenga: *Tomcat*, *Jetty* u *Undertow* porque vienen embebidos como dependencias y simplemente se agrega el que se adapte a las necesidades. Todo esto con el gestor de dependencias que se elija, bien sea *Maven* o *Gradle*, sin tocar un servidor o configurar otro tipo de cosas.
- **Inversión de control e inyección de dependencias:** En el contexto de Spring tenemos dos conceptos muy importantes: la *Inyección de dependencias* y la *inversión de control*.
Aunque son 2 conceptos que se relacionan, son distintos. Es decir, la *inyección de dependencias* es un patrón de diseño como *singleton*, *prototype*, *builder*, *observer*, etc. y permite implementar el principio de *inversión de control*
- **Arquitectura REST en Spring Boot y estereotipos:** *Spring Boot* al hacer parte de toda la arquitectura de *Spring* puede interactuar con los demás proyectos, entre ellos tenemos *Spring Framework*, proyecto que provee soporte para construir aplicaciones web, principalmente se pueden crear API propias y desplegar servicios *REST* propios para que se pueda interactuar con otros servicios. Incluso para que se desplieguen de tal forma que cualquier cliente los consuma, bien sea una aplicación móvil, una aplicación web, o cualquier otro tipo de cliente que pueda conectarse bajo el protocolo HTTP.

Toda esta arquitectura se integra desde *Spring Boot* como foco principal, pudiéndose además hacer uso de **anotaciones** tales como *Repository*, *Service*, *Componente*, entre muchas otras que permiten desarrollar una aplicación bajo la arquitectura que provee el framework de *Spring*, bien sea

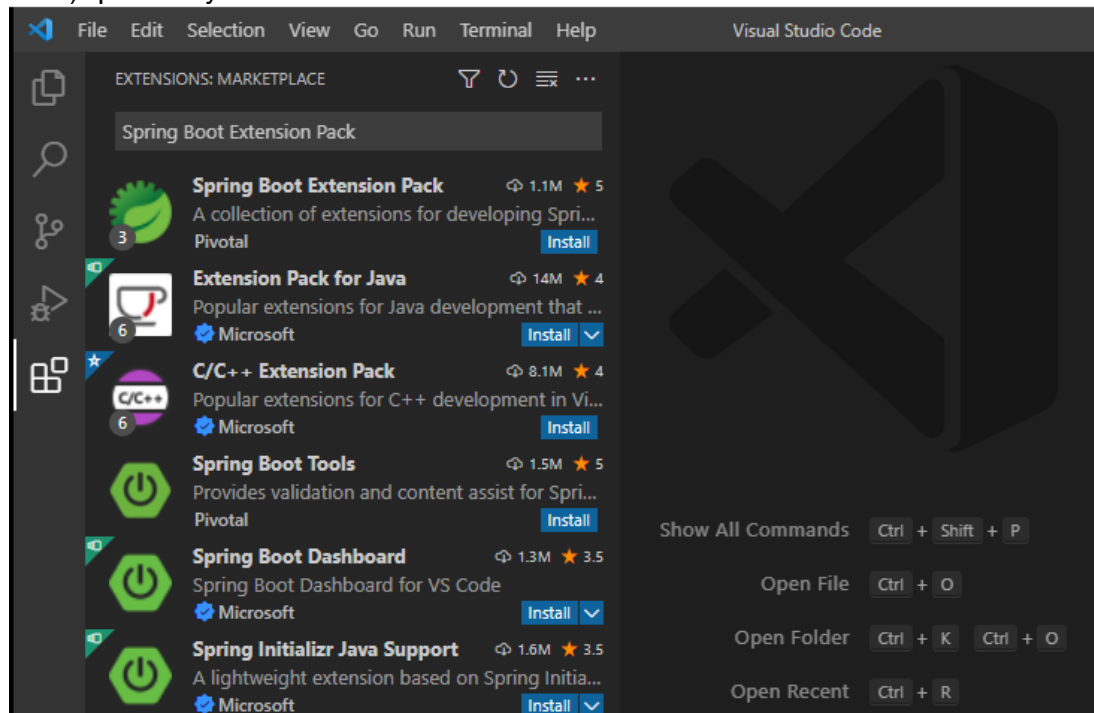
para interactuar con bases de datos, crear lógica de negocio o simplemente desarrollar un componente general para todas las capas de la aplicación.

Spring Boot en Visual Studio Code

Visual Studio Code (VS Code) es un entorno de desarrollo liviano ideal para los desarrolladores de aplicaciones *Spring Boot* y hay varias extensiones útiles de VS Code que incluyen:

- Herramientas *Spring Boot*
- Spring Initializr
- Tablero de *Spring Boot*


Se recomienda instalar *Spring Boot Extension Pack* (Paquete de extensión *Spring Boot*) que incluye todas las extensiones anteriores.



Para desarrollar una aplicación *Spring Boot* en VS Code, se debe instalar lo siguiente:

- Kit de desarrollo de Java (JDK)
- Paquete de extensión para Java
- Paquete de extensión *Spring Boot*

Instalando el Paquete de extensión para Java:




Extension Pack for Java

v0.25.0 Preview


Microsoft | 14,047,499 | ★★★★★ (51)


Popular extensions for Java development that provides Java IntelliSense, debugging, testing, Maven/Gradle support, project management and more


Installing 


[Details](#) [Feature Contributions](#) [Changelog](#)

Extension Pack (6)

**IntelliCode**
AI-assisted development
Microsoft Installing

**Language Support for Java(TM) by Red Hat**
Java Linting, Intellisense, formatting, refactor...
Red Hat Installing

**Debugger for Java**
A lightweight Java debugger for Visual Studi...
Microsoft Installing

**Maven for Java**
Manage Maven projects, execute goals, gen...
Microsoft Installing


Extension Pack for Java

Extension Pack for Java is a collection of popular extensions that can help write, test and debug Java applications in Visual Studio Code. Check out [Java in VS Code](#) to get started.

Se debe descargar e instalar la versión de JDK compatible:

Get Started

< Get Started



Get Started with Java Development

Your first steps to set up powerful Java tools in a lightweight, performant editor!

☒ **Get your runtime ready**

The Extension Pack for Java requires at least one Java runtime to be installed.

Install JDK

☐ Explore your project

☐ Launch, debug and test

☐ Extensions for additional tools and framew..

☐ Explore more Java resources

✓ Mark Done

Install JDK

If you don't have JDK installed on your machine, you can install it by clicking on **Install JDK**.

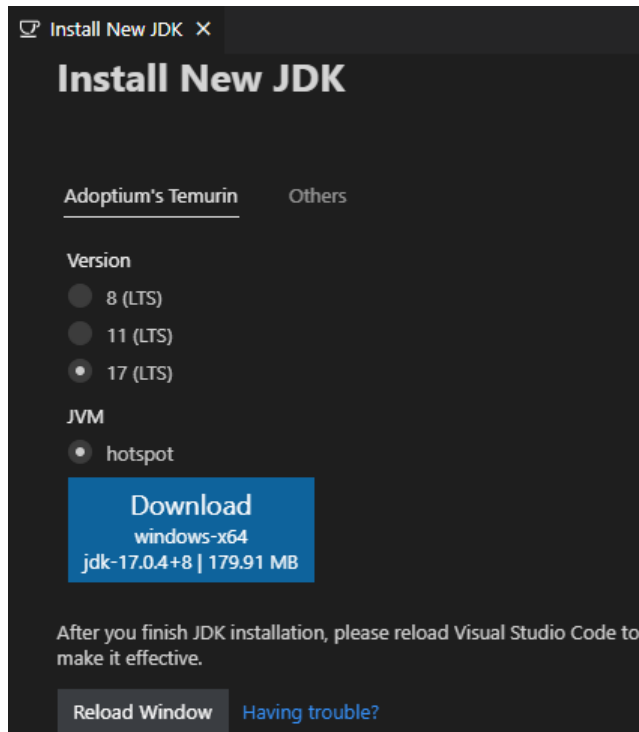
To verify it's installed, [create a new terminal](#) and try running the following command:

```
java -version
```

You should see something similar to the following:

```
java version "1.8.0_311"  
Java(TM) SE Runtime Environment (build 1.8.0_311-b11)  
Java HotSpot(TM) 64-Bit Server VM (build 25.311-b11, mixed
```

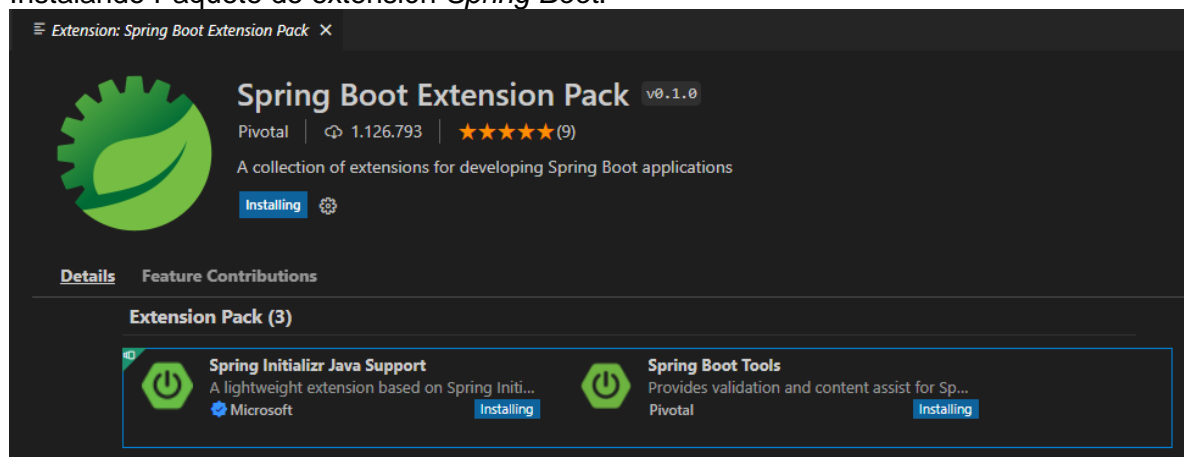
The detailed output will be based on the JDK you install.



Una vez instalado se debe recargar la ventana de VS Code y verificar que esté instalado el JDK:



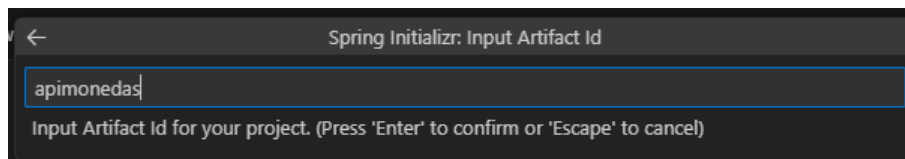
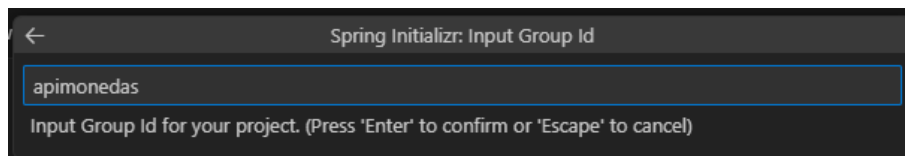
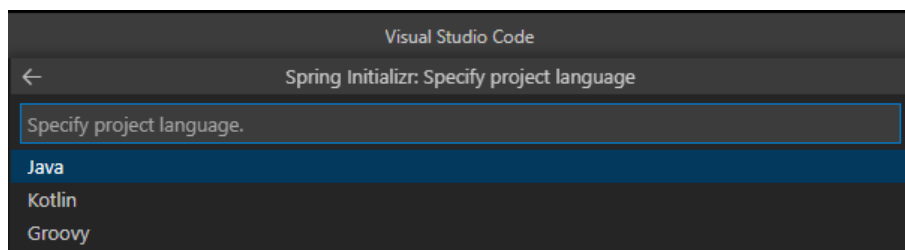
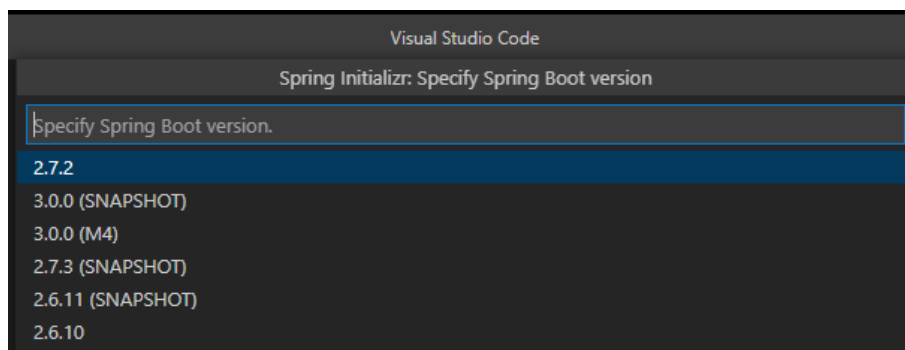
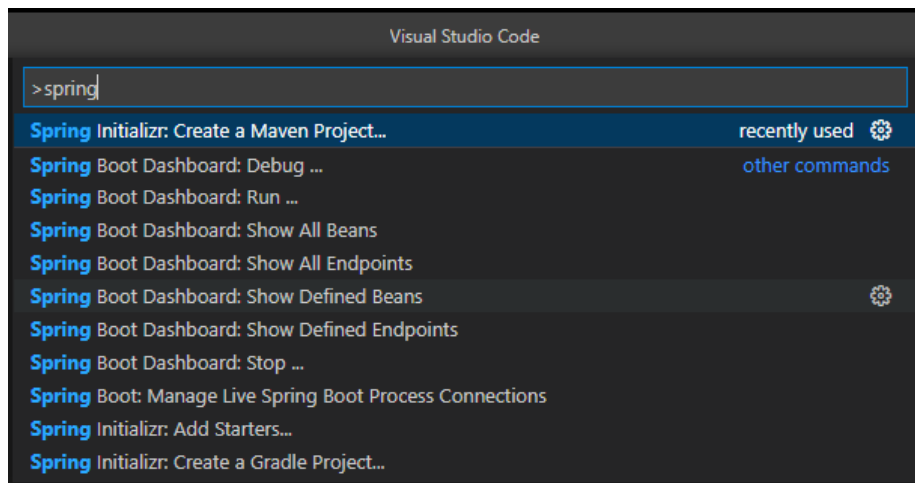
Instalando Paquete de extensión *Spring Boot*:

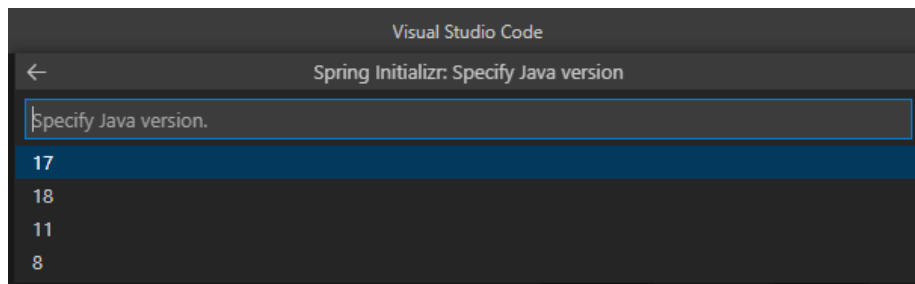
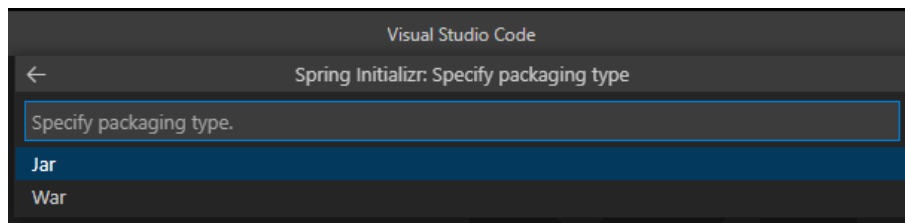


Una vez instaladas las anteriores extensiones, se puede proceder a crear un proyecto.

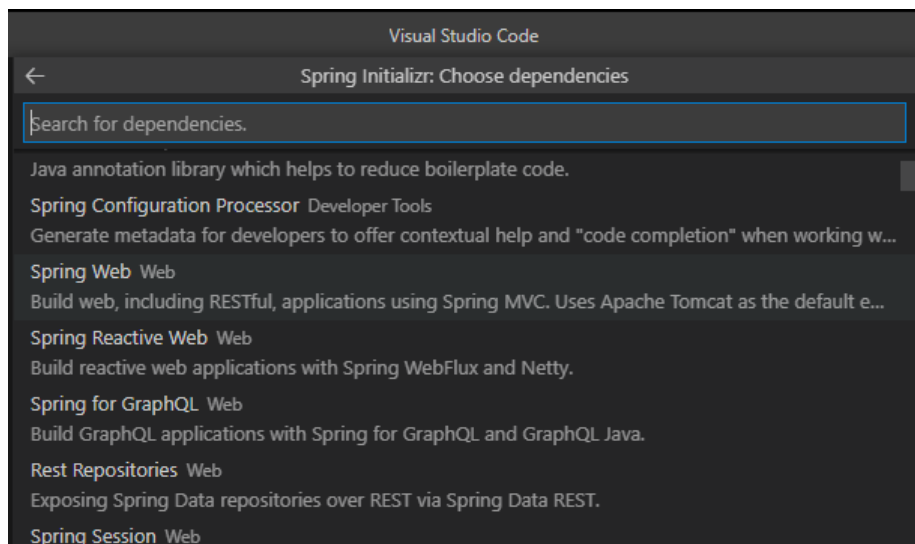
La extensión *Spring Initializr* le permite buscar dependencias y generar nuevos proyectos *Spring Boot*.

Se debe abrir la paleta de comandos (Ctrl+Shift+P) y escribir *Spring Initializr* para comenzar a generar un proyecto *Maven* o *Gradle* y luego seguir el asistente:

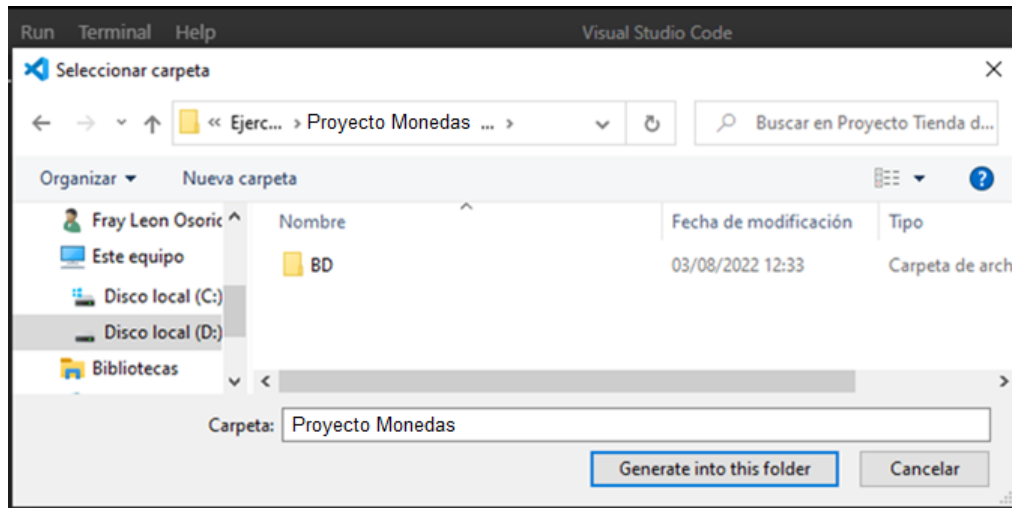




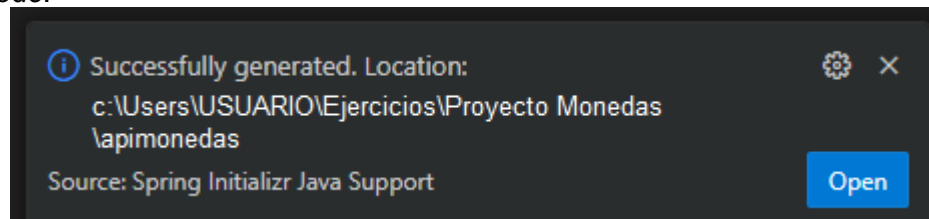
Cuando se eligen las dependencias, en el caso de un proyecto *API REST* se debe elegir **Spring Web**:



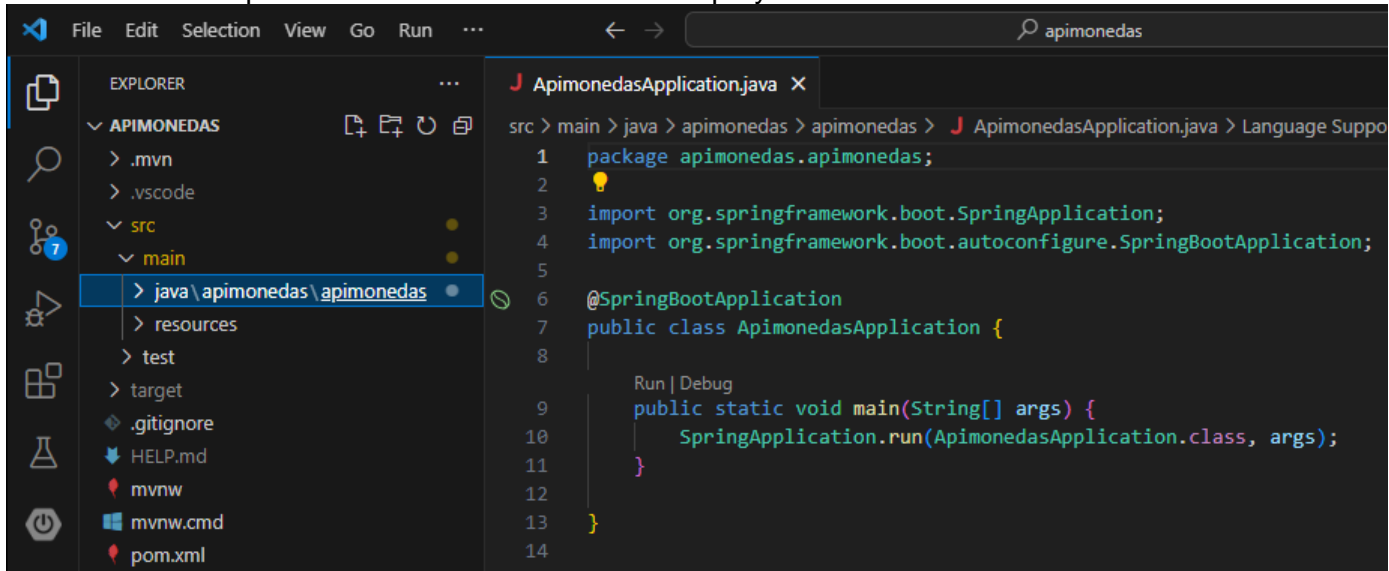
El asistente termina con la definición de la ruta donde se alojará el proyecto:



Se procede por tanto a generar el proyecto en la carpeta, y cuando termine, sale una ventana de mensaje con el proceso de generación exitoso y un botón para abrirlo en VS Code:

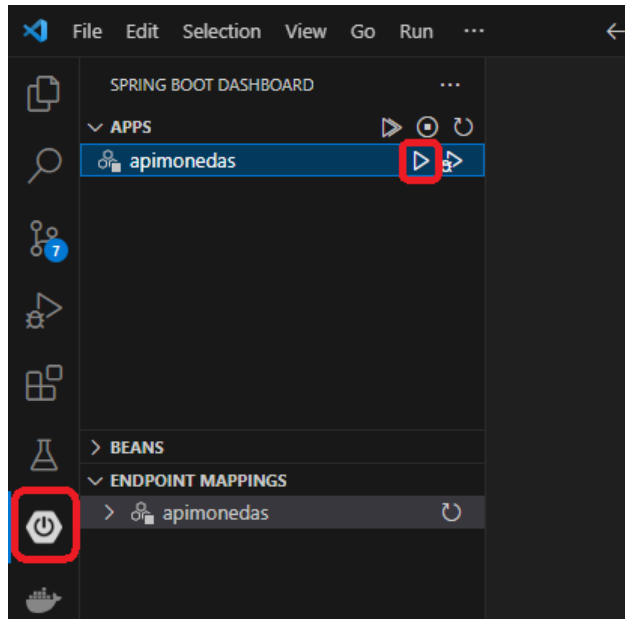


El ambiente para continuar con el desarrollo del proyecto luciría así:

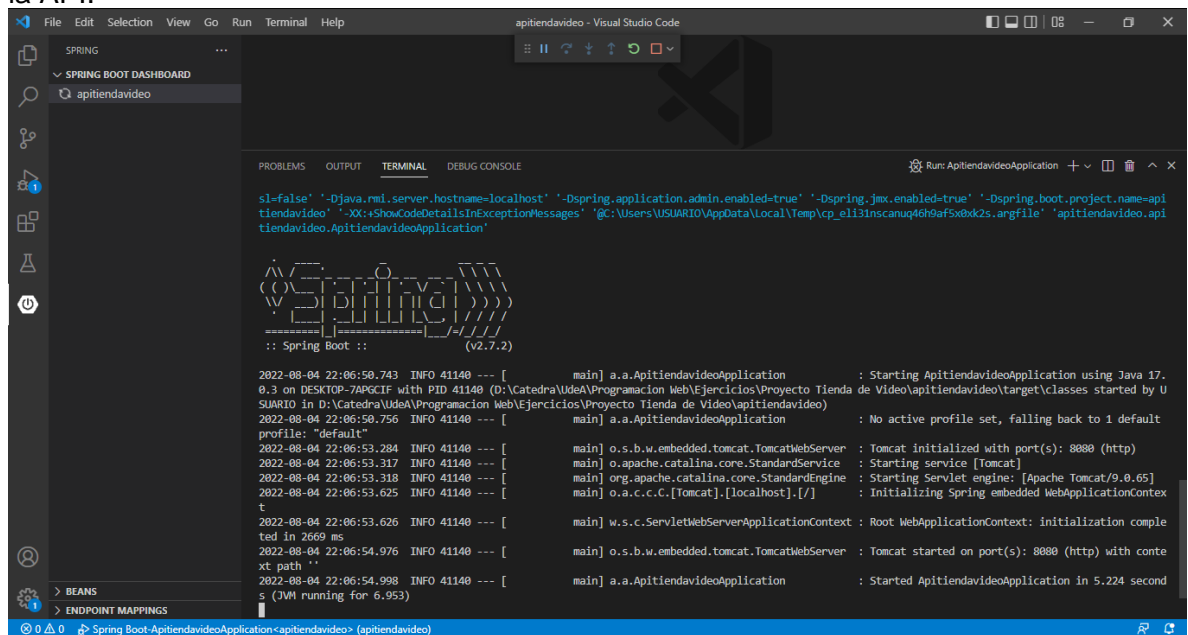


Ejecutar la aplicación

Además de usar F5 para ejecutar la aplicación, existe la extensión *Spring Boot Dashboard*, que permite ver y administrar todos los proyectos *Spring Boot* disponibles en el espacio de trabajo, así como iniciar, detener o depurar rápidamente su proyecto.



Al hacer clic en el botón *Ejecutar* se desencadena un proceso que deja ejecutándose la API:



¿Qué es PostgreSQL?

PostgreSQL, o simplemente Postgres, es un sistema de código abierto de administración de bases de datos del tipo relacional, aunque también es posible ejecutar consultas que sean no relaciones. En este sistema, las consultas relacionales se basan en *SQL*, mientras que las no relacionales hacen uso de *JSON*.

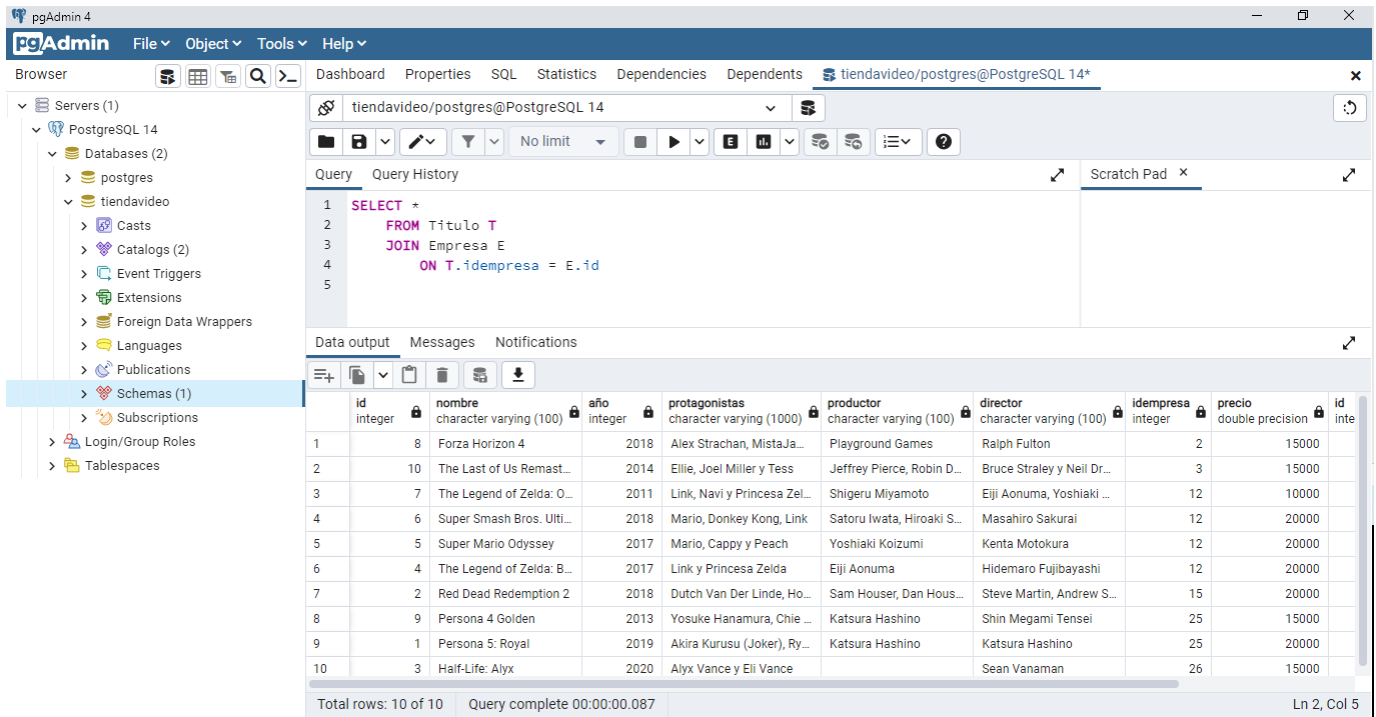
Además de ser un sistema de código abierto es gratuito y multiplataforma, y su desarrollo es llevado adelante por una gran comunidad de colaboradores de todo el mundo que día a día ponen su granito de arena para hacer de este sistema una de las opciones más sólidas a nivel de bases de datos.

Dos detalles a destacar de PostgreSQL es que posee **data types** (*tipos de datos*) avanzados y permite ejecutar optimizaciones de rendimiento avanzadas, que son características que por lo general solo se ven en sistemas de bases de datos comerciales, como por ejemplo SQL Server de Microsoft u Oracle de la compañía homónima.

Una característica extremadamente importante de *PostgreSQL* es su gran capacidad para el manejo de grandes volúmenes de datos, algo en lo que otros sistemas como *MySQL* aún no hacen tan bien. Las bases de datos de gran tamaño pueden hacer pleno uso del **MVCC (Multi-Version Concurrency Control)** de *PostgreSQL*, resultando en un gran rendimiento. MVCC es un método de control que permite realizar tareas de escritura y lectura simultáneamente.

Otro punto muy importante que no se debe dejar de lado es el cumplimiento de *ACID*. ¿Qué es ACID? Estas siglas en inglés refieren a: **atomicity, consistency, isolation y durability**, (atomicidad, consistencia, aislamiento y durabilidad) de las transacciones que se realizan en una base de datos. ¿Y por qué es tan importante? Porque tener soporte completo de *ACID* da la seguridad de que, si se produce una falla durante una transacción, los datos no se perderán ni terminarán donde no deban.

La administración de *PostgreSQL* se vuelve muy sencilla por medio de aplicativos como **PgAdmin**, que básicamente viene a ser un *phpMyAdmin* orientado para *PostgreSQL*. La posibilidad de realizar diversos procedimientos en forma sencilla hace que *PgAdmin* sea ampliamente utilizado, aunque también permite realizar tareas más complejas, así que tanto novatos como usuarios expertos hacen uso de él.



The screenshot shows the pgAdmin 4 interface. The left sidebar displays the database structure, including Servers, Databases, and Schemas. The main pane shows a query window with the following SQL query:

```

1 SELECT *
2 FROM Titulo T
3 JOIN Empresa E
4 ON T.idempresa = E.id
5

```

The query results are displayed in a table with the following columns: id, nombre, año, protagonistas, productor, director, idempresa, precio, and id. The table contains 10 rows of data.

id	nombre	año	protagonistas	productor	director	idempresa	precio	id
1	Forza Horizon 4	2018	Alex Strachan, MistaJa...	Playground Games	Ralph Fulton	2	15000	
2	The Last of Us Remast...	2014	Ellie, Joel Miller y Tess	Jeffrey Pierce, Robin D...	Bruce Straley y Neil Dr...	3	15000	
3	The Legend of Zelda: O...	2011	Link, Navi y Princesa Zel...	Shigeru Miyamoto	Eiji Aonuma, Yoshiaki ...	12	10000	
4	Super Smash Bros. Ulti...	2018	Mario, Donkey Kong, Link	Satoru Iwata, Hiroaki S...	Masahiro Sakurai	12	20000	
5	Super Mario Odyssey	2017	Mario, Cappy y Peach	Yoshiaki Koizumi	Kenta Motokura	12	20000	
6	The Legend of Zelda: B...	2017	Link y Princesa Zelda	Eiji Aonuma	Hidemaro Fujibayashi	12	20000	
7	Red Dead Redemption 2	2018	Dutch Van Der Linde, Ho...	Sam Houser, Dan Hous...	Steve Martin, Andrew S...	15	20000	
8	Persona 4 Golden	2013	Yosuke Hanamura, Chie ...	Katsura Hashino	Shin Megami Tensei	25	15000	
9	Persona 5: Royal	2019	Akira Kurusu (Joker), Ry...	Katsura Hashino	Katsura Hashino	25	20000	
10	Half-Life: Alyx	2020	Alyx Vance y Eli Vance		Sean Vanaman	26	15000	

Total rows: 10 of 10 Query complete 00:00:00.087 Ln 2, Col 5

¿Qué es JPA?

Casi cualquier aplicación que vayamos a construir, tarde o temprano tiene que lidiar con la **persistencia de sus datos**. Es decir, debemos lograr que los datos que maneja o genera la aplicación se almacenen fuera de esta para su uso posterior.

Esto, por regla general, implica el uso de un sistema de base de datos, bien sea el tradicional modelo relacional basado en SQL, o bien de el de tipo documental (también conocido como No-SQL).

Por tanto a la hora de desarrollar, los programas modernos modelan su información utilizando clases (de la Programación Orientada a Objetos), pero las bases de datos utilizan otros paradigmas: las relaciones (también llamadas a veces "Tablas") y las claves externas que las relacionan. Esta diferencia entre la forma de programar y la forma de almacenar da lugar a lo que se llama "desfase de impedancia" y complica la persistencia de los objetos.

Para minimizar ese desfase y facilitar a los programadores la persistencia de sus objetos de manera transparente, nacen los denominados **Mapeadores Objeto-Relacionales (ORM)** por su sigla en inglés **Object Relational Mapping**.

Con lo anterior en mente, se puede definir entonces que es la **API de Persistencia de Java o JPA**.

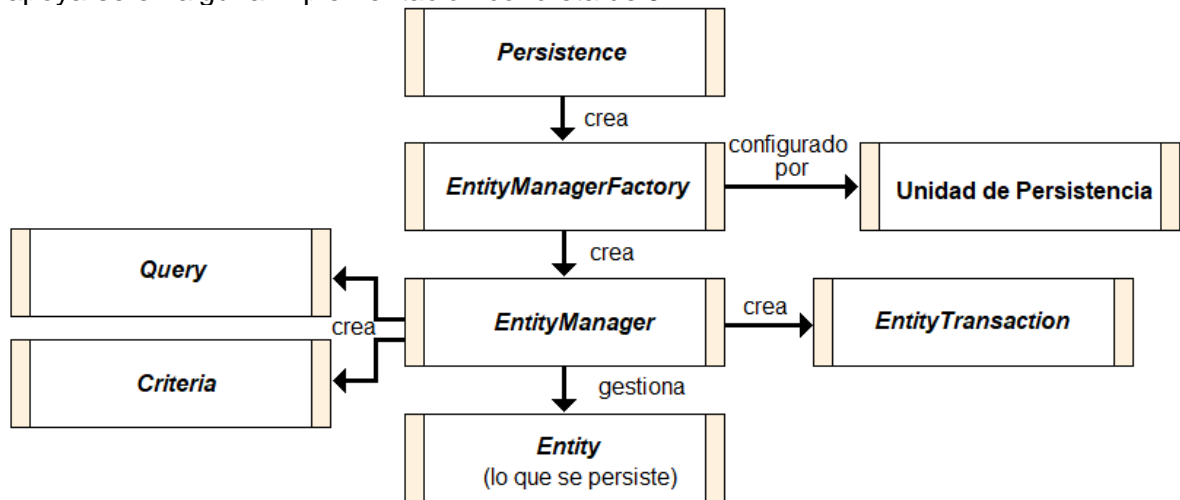
JPA es una especificación que indica cómo se debe realizar la persistencia (almacenamiento) de los objetos en programas Java. Es clave entender que se usa la palabra "Especificación" porque JPA no tiene una implementación concreta, sino

que, como se verá enseguida, existen diversas tecnologías que implementan JPA para darle concreción.

Aunque forma parte de Java empresarial, las implementaciones de JPA se pueden emplear en cualquier tipo de aplicación aislada, sin necesidad de usar ningún servidor de aplicaciones, como una mera biblioteca de acceso a datos.

¿Cómo funciona JPA?

Dado que es una especificación, JPA no proporciona clase alguna para poder trabajar con la información. Lo que hace es proveer de una serie de interfaces que se pueden utilizar para implementar la capa de persistencia de una aplicación, permitiendo apoyarse en alguna implementación concreta de JPA.



Es decir, en la práctica significa que lo que se va a utilizar es una biblioteca de persistencia que implementa JPA, no JPA directamente.

Existen diversas implementaciones disponibles, como *DataNucleus*, *ObjectDB*, o *Apache OpenJPA*, pero las dos más utilizadas son *EclipseLink* y sobre todo **Hibernate**.

Hibernate en la actualidad es casi el "estándar" de facto, puesto que es la más utilizada, sobre todo en las empresas. Es tan popular que existen hasta versiones para otras plataformas, como **NHibernate** para la plataforma .NET de microsoft. Es un proyecto muy maduro (de hecho, la especificación JPA original partió de él), muy bien documentado y que tiene un gran rendimiento.

Entre las principales características se tiene:

- **Mapeado de Entidades.** Lo primero que se tiene que hacer para usar una implementación de JPA es añadir la dependencia al proyecto y configurar el sistema de persistencia (por ejemplo, la cadena de conexión a una base de datos). Pero, tras esas cuestiones de "carpintería", lo más básico que se deberá hacer y que es el núcleo de la especificación es el mapeado de entidades.

El "mapeado" se refiere a definir cómo se relacionan las clases de la aplicación con los elementos del sistema de almacenamiento.

Si se considera el caso común de acceso a una base de datos relacional, sería definir:

- La relación existente entre las clases de la aplicación y las tablas de la base de datos
- La relación entre las propiedades de las clases y los campos de las tablas
- La relación entre diferentes clases y las claves externas de las tablas de la base de datos

Este es un ejemplo de este tipo de mapeo:

```
import org.hibernate.annotations.GenericGenerator;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.Table;

@Entity
@Table(name = "pais")
public class Pais {

    @Id
    @Column(name = "id")
    @GeneratedValue(strategy = GenerationType.AUTO, generator =
"secuencia_pais")
    @GenericGenerator(name = "secuencia_pais", strategy =
"increment")
    private long id;

    @Column(name = "pais", length = 100, unique = true)
    private String nombre;

    @Column(name = "codigoalfa2")
    private String codigoAlfa2;

    @Column(name = "codigoalfa3")
    private String codigoAlfa3;

    @ManyToOne
    @JoinColumn(name = "idmoneda", referencedColumnName = "id")
    private Moneda moneda;

}
```


Como se puede observar en este código, lo que define JPA es también una serie de anotaciones con las que se pueden decorar las clases, propiedades y métodos para indicar esos "mapeados":

@Entity		
@Table		
@Id		
@GeneratedValue		
@GenericGenerator		
@Column		
@ManyToOne		
@JoinColumn		

Además, JPA simplifica aún más el trabajo gracias al uso de una serie de convenciones por defecto que sirven para un alto número de casos de uso habituales. Por ello, solo deberemos anotar aquellos comportamientos que queramos modificar o aquellos que no se pueden deducir de las propias clases. Por ejemplo, aunque existe una anotación (@Table) para indicar el nombre de la tabla en la base de datos que está asociada a una clase, por defecto si no indicamos otra cosa se considerará que ambos nombres coinciden. Por ejemplo, en el fragmento anterior, al haber anotado con @Entity la clase Factura se considera automáticamente que la tabla en la base de datos donde se guardan los datos de las facturas se llama también Factura. Pero podríamos cambiarla poniéndole la anotación @Table(name="Invoices"), por ejemplo.

Ejemplos de conexión de *Spring Boot* a la base de datos *PostgreSQL*

A continuación, se expone como configurar y escribir código para conectarse a un servidor de base de datos *PostgreSQL* en una aplicación *Spring Boot*. Las dos formas comunes son:

- Usar Spring JDBC con *JdbcTemplate*
- Usar Spring Data JPA

Para conectar una aplicación *Spring Boot* a una base de datos *PostgreSQL*, se debe seguir estos pasos:

- Agregar una dependencia para el controlador JDBC de *PostgreSQL*, que es necesario para permitir que las aplicaciones de Java puedan comunicarse con un servidor de base de datos de *PostgreSQL*
- Configurar las propiedades del origen de datos para la información de conexión de la base de datos
- Agregar una dependencia para *Spring JDBC* o *Spring Data JPA*, según la necesidad:
 - Usar *Spring JDBC* para ejecutar declaraciones de SQL simples
 - Usar *Spring Data JPA* para un uso más avanzado, como asignar clases de Java a tablas y objetos de Java a filas, y aprovechar las ventajas de la API *Spring Data JPA*.

A continuación, se muestran los detalles de configuración y ejemplos de código.

1. Agregar dependencia para el controlador JDBC de PostgreSQL

Se declara la siguiente dependencia en el archivo *pom.xml* del proyecto:

```
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <scope>runtime</scope>
</dependency>
```

Esto utilizará la versión predeterminada especificada por *Spring Boot*.

2. Configurar las propiedades de la fuente de datos

A continuación, se debe especificar cierta información de conexión a la base de datos en el archivo de configuración de la aplicación *Spring Boot* (*application.properties*) de la siguiente manera:

```
spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.url=jdbc:postgresql://localhost:5432/monedas
spring.datasource.username=postgres
spring.datasource.password=sa
```

Aquí, la URL de JDBC apunta a un servidor de base de datos PostgreSQL que se ejecuta en localhost. Actualice la URL, el nombre de usuario y la contraseña de JDBC según el entorno.

3. Conectarse a la base de datos PostgreSQL con Spring JDBC

En el caso más simple, se puede usar *Spring JDBC* con *JdbcTemplate* para trabajar con una base de datos relacional. Para ello se debe agregar la siguiente dependencia al archivo de proyecto Maven:

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

Y el siguiente ejemplo de código es de un programa de consola *Spring Boot* que usa *JdbcTemplate* para ejecutar una declaración SQL *Insert*:

```
package paquete.ejemplo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.jdbc.core.JdbcTemplate;

@SpringBootApplication
public class EjemploApplication implements CommandLineRunner {

    @Autowired
    private JdbcTemplate jdbcTemplate;
```

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

@Override
public void run(String... args) throws Exception {
    String sql = "INSERT INTO Moneda (moneda, sigla, símbolo,
emisor) VALUES('Peso Colombiano', 'COP', '$', 'Banco de
República')";
    int rows = jdbcTemplate.update(sql);
    if (rows > 0) {
        System.out.println("Se agregó una nueva moneda.");
    }
}
}
```

Este programa insertará un nuevo registro en la tabla de *Tercero* en la base de datos *PostgreSQL*, utilizando *Spring JDBC*, que es una API delgada construida sobre JDBC.

4. Conectarse a la base de datos PostgreSQL con Spring Data JPA

Si se desea asignar clases de Java a tablas y objetos de Java a filas y aprovechar las ventajas de un framework de **Mapeo Relacional de Objetos (Object-Relational Mapping – ORM)** en inglés) como *Hibernate*, puede usar *Spring Data JPA*. Así que se debe declarar la siguiente dependencia en el proyecto:

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

Además de la URL, el nombre de usuario y la contraseña de JDBC, también se puede especificar algunas propiedades adicionales de la siguiente manera:

```
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQL81Dialect
```

Y se necesita codificar una clase de entidad (una clase *POJO* Java) para mapear con la tabla correspondiente en la base de datos, de la siguiente manera:

```
package apimonedas.apimonedas.core.entidades;

import org.hibernate.annotations.GenericGenerator;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name="moneda")
```

```
public class Moneda {

    @Id
    @Column(name = "id")
    @GeneratedValue(strategy = GenerationType.AUTO, generator =
"secuencia_moneda")
    @GenericGenerator(name = "secuencia_moneda", strategy =
"increment")
    private long id;

    @Column(name = "moneda", length = 100, unique = true)
    private String nombre;

    @Column(name = "sigla", length = 5, unique = true)
    private String sigla;

    @Column(name = "simbolo", length = 5)
    private String simbolo;

    @Column(name = "emisor", length = 100)
    private String emisor;

    public Moneda() {
    }

    public Moneda(long id, String nombre, String sigla, String
simbolo, String emisor) {
        this.id = id;
        this.nombre = nombre;
        this.sigla = sigla;
        this.simbolo = simbolo;
        this.emisor = emisor;
    }

    public long getId() {
        return id;
    }

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

    public String getNombre() {
        return nombre;
    }

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

    public String getSigla() {
        return sigla;
    }

    public void setSigla(String sigla) {
```

```
        this.sigla = sigla;
    }

    public String getSimbolo() {
        return simbolo;
    }

    public void setSimbolo(String simbolo) {
        this.simbolo = simbolo;
    }

    public String getEmisor() {
        return emisor;
    }

    public void setEmisor(String emisor) {
        this.emisor = emisor;
    }
}
```

En este código se mapeó la tabla *Moneda*

Luego, se debe declarar una interfaz de repositorio de la siguiente manera:

```
package apimonedas.apimonedas.core.interfaces.repositorios;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;

import java.util.List;

import apimonedas.apimonedas.core.entidades.Moneda;

@Repository
public interface IMonedaRepositorio extends JpaRepository<Moneda, Long> {

    @Query("SELECT m FROM Moneda m WHERE m.nombre LIKE '%' || ?1 || '%'")
    List<Moneda> buscar(String nombre);

    @Query("SELECT m FROM Pais p JOIN p.moneda m WHERE p.nombre=?1")
    Moneda buscarPorPais(String nombre);
}
```

Y luego se puede usar este repositorio en un servicio *Spring* de la siguiente manera:

```
package apimonedas.apimonedas.aplicacion;

import java.util.Date;
import java.util.List;
import java.util.Optional;

import org.springframework.stereotype.Service;
```

```
import apimonedas.apimonedas.core.entidades.CambioMoneda;
import apimonedas.apimonedas.core.entidades.Moneda;
import
apimonedas.apimonedas.core.interfaces.repositorios.ICambioMonedaR
epositorio;
import
apimonedas.apimonedas.core.interfaces.repositorios.IMonedaReposit
orio;
import
apimonedas.apimonedas.core.interfaces.servicios.IMonedaServicio;

@Service
public class MonedaServicio implements IMonedaServicio {

    private IMonedaRepositorio repositorio;
    private ICambioMonedaRepositorio repositorioCambio;

    public MonedaServicio(IMonedaRepositorio repositorio,
        ICambioMonedaRepositorio repositorioCambio) {
        this.repositorio = repositorio;
        this.repositorioCambio = repositorioCambio;
    }

    @Override
    public List<Moneda> listar() {
        return repositorio.findAll();
    }

    @Override
    public Moneda obtener(Long id) {
        var moneda = repositorio.findById(id);
        return moneda.isEmpty() ? null : moneda.get();
    }

    @Override
    public List<Moneda> buscar(String nombre) {
        return repositorio.buscar(nombre);
    }

    @Override
    public Moneda buscarPorPais(String nombre) {
        return repositorio.buscarPorPais(nombre);
    }

    public Moneda agregar(Moneda moneda) {
        moneda.setId(0);
        return repositorio.save(moneda);
    }

    @Override
    public Moneda modificar(Moneda moneda) {
        Optional<Moneda> monedaEncontrado =
repositorio.findById(moneda.getId());
        if (!monedaEncontrado.isEmpty()) {
```

```
        return repositorio.save(moneda);
    } else {
        return null;
    }
}

@Override
public boolean eliminar(Long id) {
    try {
        repositorio.deleteById(id);
        return true;
    } catch (Exception ex) {
        return false;
    }
}

@Override
public List<CambioMoneda> listarPorPeriodo(long idMoneda, Date
fecha1, Date fecha2) {
    return repositorioCambio.listarPorPeriodo(idMoneda,
fecha1, fecha2);
}
}
```

En el cual se define toda la funcionalidad tipo CRUD para operar con la tabla *Moneda* de la base de datos.

Se puede observar que también llama a otro repositorio, en este caso el encargado de operar con la tabla *CambioMoneda* para usar un funcionalidad que lista los cambios de una moneda durante un período de tiempo. Este sería el repositorio respectivo:

```
package apimonedas.apimonedas.core.interfaces.repositorios;

import java.util.Date;
import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;

import apimonedas.apimonedas.core.entidades.CambioMoneda;

@Repository
public interface ICambioMonedaRepositorio extends
JpaRepository<CambioMoneda, Long> {

    @Query("SELECT cm FROM CambioMoneda cm WHERE cm.moneda.id=?1
AND cm.fecha >= ?2 AND cm.fecha <= ?3")
    public List<CambioMoneda> listarPorPeriodo(long idMoneda, Date
fecha1, Date fecha2);
}
```

El anterior servicio implementa una interfaz denominada *IMonedaServicio* cuyo código es el siguiente

```
package apimonedas.apimonedas.core.interfaces.servicios;

import java.util.Date;
import java.util.List;

import apimonedas.apimonedas.core.entidades.CambioMoneda;
import apimonedas.apimonedas.core.entidades.Moneda;

public interface IMonedaServicio {

    public List<Moneda> listar();

    public Moneda obtener(Long id);

    public List<Moneda> buscar(String nombre);

    public Moneda buscarPorPais(String nombre);

    public Moneda agregar(Moneda moneda);

    public Moneda modificar(Moneda moneda);

    public boolean eliminar(Long id);

    public List<CambioMoneda> listarPorPeriodo(long idMoneda, Date
fecha1, Date fecha2);
}
```

Como se puede observar, es un contrato que resume la funcionalidad expuesta. El objetivo de definir primero esta interfaz para luego ser implementada en una clase externa, es mantener una separación clara y una dependencia unidireccional entre las capas de la aplicación. Esto significa que las capas internas no deben conocer los detalles de las capas externas, mientras que las capas externas pueden depender de las capas internas. Al seguir este principio, se busca lograr un diseño modular y flexible que facilite la comprensión, el mantenimiento y la evolución de la aplicación a lo largo del tiempo.

En el contexto de crear primero una interfaz y luego su implementación para un servicio en la capa de negocio, esto se alinea con el **principio de inversión de dependencias** (*Dependency Inversion Principle*) de SOLID, que es fundamental en la arquitectura de cebolla. Este principio establece que:

- Las dependencias deben ser hacia abstracciones, no implementaciones concretas.
- Los detalles de bajo nivel no deben depender de los detalles de alto nivel. Ambos deben depender de abstracciones.

Siguiendo este principio, al crear primero la interfaz (una abstracción) para un servicio en la capa de negocio, se establece un contrato claro entre la capa de negocio y las capas externas que la utilizan. Las capas externas pueden depender de esta interfaz sin necesidad de conocer los detalles de su implementación concreta.

Esto permite lograr varios beneficios:

- Flexibilidad: Se puede cambiar la implementación del servicio sin afectar a las capas externas, siempre y cuando se mantenga la interfaz.
- Pruebas: Facilita la escritura de pruebas unitarias y la realización de pruebas de integración utilizando mocks o stubs de la interfaz.
- Desacoplamiento: Reduce el acoplamiento entre las capas de la aplicación al eliminar las dependencias directas entre las implementaciones concretas.

Para terminar, se puede usar el anterior servicio en un controlador *Spring MVC* o clase empresarial de la siguiente manera:

```
package apimonedas.apimonedas.presentacion;

import java.util.List;

import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import apimonedas.apimonedas.core.DTOs.PeriodoDto;
import apimonedas.apimonedas.core.entidades.CambioMoneda;
import apimonedas.apimonedas.core.entidades.Moneda;
import apimonedas.apimonedas.core.interfaces.servicios.IMonedaServicio;

@RestController
@RequestMapping("/api/monedas")
public class MonedaControlador {
    private IMonedaServicio servicio;

    public MonedaControlador(IMonedaServicio servicio) {
        this.servicio = servicio;
    }

    @RequestMapping(value = "/listar", method = RequestMethod.GET)
    public List<Moneda> listar() {
        return servicio.listar();
    }

    @RequestMapping(value = "/obtener/{id}", method =
RequestMethod.GET)
    public Moneda obtener(@PathVariable long id) {
        return servicio.obtener(id);
    }

    @RequestMapping(value = "/buscar/{nombre}", method =
RequestMethod.GET)
    public List<Moneda> buscar(@PathVariable String nombre) {
        return servicio.buscar(nombre);
    }
}
```

```
@RequestMapping(value = "/buscarporpais/{nombre}", method =
RequestMethod.GET)
public Moneda buscarPorPais(@PathVariable String nombre) {
    return servicio.buscarPorPais(nombre);
}

@RequestMapping(value = "/agregar", method =
RequestMethod.POST)
public Moneda crear(@RequestBody Moneda moneda) {
    return servicio.agregar(moneda);
}

@RequestMapping(value = "/modificar", method =
RequestMethod.PUT)
public Moneda actualizar(@RequestBody Moneda moneda) {
    return servicio.modificar(moneda);
}

@RequestMapping(value = "/eliminar/{id}", method =
RequestMethod.DELETE)
public boolean eliminar(@PathVariable long id) {
    return servicio.eliminar(id);
}

@RequestMapping(value = "/listarporperiodo", method =
RequestMethod.GET)
public List<CambioMoneda> listarPorPeriodo(@RequestBody
PeriodoDto periodo) {
    System.out.println("periodo="+periodo.getDesde()+"
"+periodo.getHasta()+" idmoneda="+periodo.getIdMoneda());

    return servicio.listarPorPeriodo(periodo.getIdMoneda(),
periodo.getDesde(), periodo.getHasta());
}
}
```

En este código se plantean funcionalidad tipo CRUD sobre la tabla *Moneda* y una consulta sobre la tabla relacionada *CambioMoneda*, todos ellos mediante métodos *REST* definido por las rutas:

Listar todos registros	/api/monedas/listar
Obtener el registro de una moneda basado en la clave primaria	/api/monedas/obtener/{id}
Buscar el registro de la moneda de un país	/api/monedas/buscarporpais/{nombre}
Agregar un registro moneda	/api/monedas/agregar
Modificar un registro existente de moneda	/api/monedas/modificar
Eliminar un registro de moneda	/api/monedas/eliminar/{id}

Listar los registros de cambios de moneda durante un período de tiempo	/api/monedas/listarporperiodo
--	---

En estas rutas se puede observar el paso de parámetros (mediante variables encerradas entre comillas). Pero algunas requieren en el *request* (solicitud) especificar objetos en formato *JSON*, como en el caso de agregar o modificar:

```
{
  "id": 35,
  "nombre": "Peso colombiano",
  "sigla": "COP",
  "simbolo": "$",
  "emisor": "El Banco de la República"
}
```

O el de listar los cambios:

```
{
  "desde": "2018-01-01",
  "hasta": "2018-06-01",
  "idMoneda": 35
}
```

¿Qué es y para qué sirve Maven?

Gradle y **Maven**, en resumen, son automatizadores de tareas principalmente utilizados para compilar proyectos en Java. Es por esta razón que, explorando un archivo **pom.xml** (en *Maven*) o un **build.gradle** (en *Gradle*), se pueden encontrar plugins de Java con nombres como “build”

Maven es una herramienta de software para la gestión y construcción de proyectos Java creada por Jason van Zyl, de Sonatype, en 2002. Es similar en funcionalidad a *Apache Ant* (y en menor medida a *PEAR* de *PHP* y *CPAN* de *Perl*), pero tiene un modelo de configuración de construcción más simple, basado en un formato XML.

¿Qué es el archivo pom.xml?

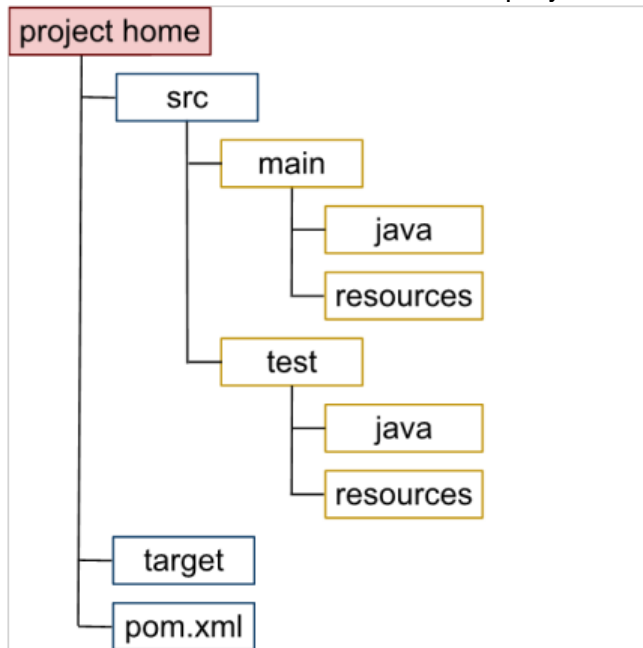
La unidad básica de trabajo en Maven es el llamado **Modelo de Objetos de Proyecto** conocido simplemente como POM (**Project Object Model** en inglés).

Se trata de un archivo XML llamado *pom.xml* que se encuentra por defecto en la raíz de los proyectos y que contiene toda la información del proyecto: su configuración, sus dependencias, etc...

El hecho de utilizar un archivo XML revela su edad. Maven se creó en 2002, cuando XML era lo más usado. Si se hubiese creado unos pocos años después seguramente tendríamos un *pom.json* y si hubiese sido más reciente un *pom.yml*. Modas que vienen y van. No obstante, el formato XML, aunque engorroso, es muy útil a la hora de definir con mucho detalle cómo debe ser cada propiedad y, también, para poder comprobarlas.

Incluso, aunque nuestro proyecto, que usa Maven, tenga un archivo `pom.xml` sin opciones propias, prácticamente vacío, estará usando el modelo de objetos para definir los valores por defecto del mismo. Por ejemplo, por defecto, el directorio donde está el código fuente es **`src/main/java`**, donde se compila el proyecto es **`target`** y donde se ubican los test unitarios es en **`src/main/test`**, etc... Al `pom.xml` global, con los valores predeterminados se le llama ***Súper POM***.

Esta sería la estructura habitual de un proyecto Java que utiliza Maven:



Probando una API REST con Postman

Todo desarrollador web ha estado en la situación de gestionar APIs tanto propias de un proyecto o APIs de integración con sistemas tercerizados, las cuales han de requerir mantenimiento eficiente y rápido.

Postman en sus inicios nace como una extensión que podía ser utilizada en el navegador *Chrome* de *Google* y básicamente permite realizar peticiones de una manera simple para testear APIs de tipo REST propias o de terceros.

Gracias a los avances tecnológicos, *Postman* ha evolucionado y ha pasado de ser de una extensión a una aplicación que dispone de herramientas nativas para diversos sistemas operativos como lo son Windows, Mac y Linux.

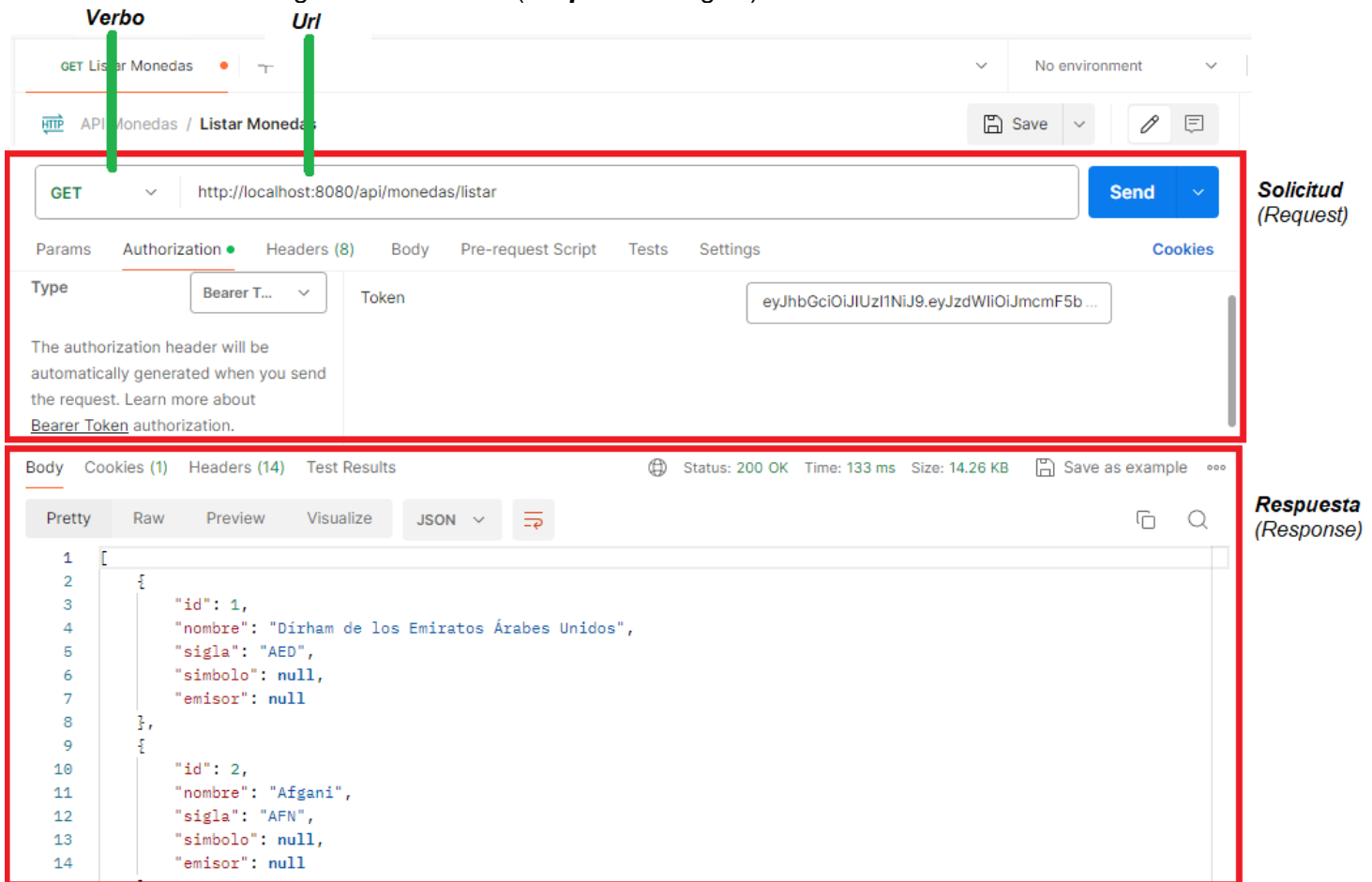
¿Para qué sirve Postman?

Postman sirve para múltiples tareas dentro de las cuales se destacan:

- Testear colecciones o catálogos de APIs tanto para Frontend como para Backend
- Organizar en carpetas, funcionalidades y módulos los servicios web
- Permite gestionar el ciclo de vida (conceptualización y definición, desarrollo, monitoreo y mantenimiento) de una API

- Generar documentación de las APIs
- Trabajar con entornos (calidad, desarrollo, producción) y de este modo es posible compartir a través de un entorno cloud, la información con el resto del equipo involucrado en el desarrollo

Por ejemplo, para probar un controlador para operar con la tabla *Moneda*, se podría realizar la siguiente **Solicitud (Request)** en inglés):



The screenshot shows a REST client interface with two main sections. The top section, labeled 'Solicitud (Request)', shows a GET request to the URL 'http://localhost:8080/api/monedas/listar'. The request is configured with a Bearer token 'eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJmcmF5b...' and is ready to be sent. The bottom section, labeled 'Respuesta (Response)', shows the JSON response in a 'Pretty' format. The response contains two objects representing currency entries: one for 'Dirham de los Emiratos Árabes Unidos' (AED) and another for 'Afgani' (AFN).

Solicitud (Request)

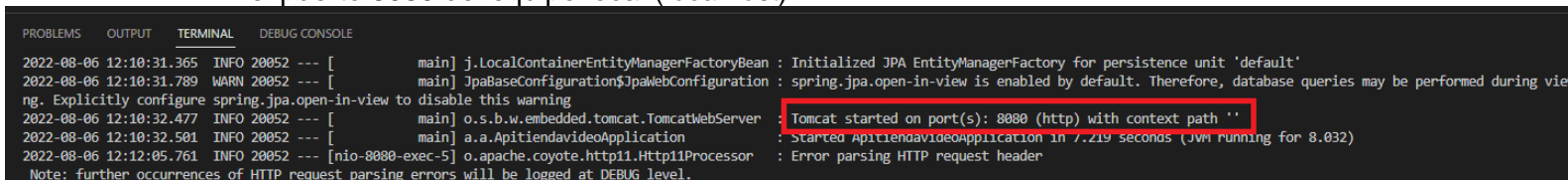
Verbo: GET
Url: http://localhost:8080/api/monedas/listar

Params: Authorization (Bearer Token)
Token: eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJmcmF5b...

Respuesta (Response)

```
[
  {
    "id": 1,
    "nombre": "Dirham de los Emiratos Árabes Unidos",
    "sigla": "AED",
    "simbolo": null,
    "emisor": null
  },
  {
    "id": 2,
    "nombre": "Afgani",
    "sigla": "AFN",
    "simbolo": null,
    "emisor": null
  }
]
```

Para poder crear la **URL** de la solicitud, es necesario comprender que el aplicativo ha sido alojado en un servidor web. Para *Spring Boot* este servidor es *Tomcat* y se utiliza el puerto 8080 del equipo local (*localhost*):



The screenshot shows a terminal window with a Java application log. The log entries show the application starting up, including the initialization of JPA and the start of the Tomcat web server. A red box highlights the line: 'Tomcat started on port(s): 8080 (http) with context path '''. This confirms that the application is running on port 8080 of the local host.

```
2022-08-06 12:10:31.365 INFO 20052 --- [main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2022-08-06 12:10:31.789 WARN 20052 --- [main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view rendering. Explicitly configure spring.jpa.open-in-view to disable this warning
2022-08-06 12:10:32.477 INFO 20052 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2022-08-06 12:10:32.581 INFO 20052 --- [main] a.a.AptiendavideoApplication : Started AptiendavideoApplication in 7.219 seconds (JVM running for 8.032)
2022-08-06 12:12:05.761 INFO 20052 --- [nio-8080-exec-5] o.apache.coyote.http11.Http11Processor : Error parsing HTTP request header
Note: further occurrences of HTTP request parsing errors will be logged at DEBUG level.
```

Ahora bien, además de especificar el equipo y el puerto, la **Url** puede tener rutas, en este caso se definieron `/api/monedas` y `/listar` (mediante la anotación `@RequestMapping` en el controlador).

En consecuencia, la Url de la solicitud queda así:

<http://localhost:8080/api/monedas/listar>

Adicionalmente, se debe especificar que es un método *GET*. Al enviar la solicitud (mediante el botón “Send”), se espera una respuesta, cuyo estado será 200 si se pudo ejecutar de manera exitosa.

La respuesta en este caso viene acompañada por un listado en *JSON*, que contiene la información solicitada:

```
[
  {
    "id": 1,
    "nombre": "Dirham de los Emiratos Árabes Unidos",
    "sigla": "AED",
    "simbolo": null,
    "emisor": null
  },
  {
    "id": 2,
    "nombre": "Afgani",
    "sigla": "AFN",
    "simbolo": null,
    "emisor": null
  },
  {
    "id": 3,
    "nombre": "Lek",
    "sigla": "ALL",
    "simbolo": null,
    "emisor": null
  },
  {
    "id": 4,
    "nombre": "Dram armenio",
    "sigla": "AMD",
    "simbolo": null,
    "emisor": null
  },
  {
    "id": 5,
    "nombre": "Florín antillano neerlandés",
    "sigla": "ANG",
    "simbolo": null,
    "emisor": null
  },
  {
    "id": 6,
    "nombre": "Kwanza",
    "sigla": "AOA",
    "simbolo": null,
    "emisor": null
  },
]
```

```
{
  "id": 7,
  "nombre": "Peso argentino",
  "sigla": "ARS",
  "simbolo": null,
  "emisor": null
},
{
  "id": 8,
  "nombre": "Dólar australiano",
  "sigla": "AUD",
  "simbolo": null,
  "emisor": null
},
{
  "id": 9,
  "nombre": "Florín arubeño",
  "sigla": "AWG",
  "simbolo": null,
  "emisor": null
},
{
  "id": 10,
  "nombre": "Manat azerbaiyano",
  "sigla": "AZN",
  "simbolo": null,
  "emisor": null
},
...
]
```

Esta es una de las maneras más simples de solicitud.

Teniendo como insumos la base de datos en *PostgreSQL* y el ambiente de desarrollo de *Visual Studio Code* con las extensiones para *Java* y *Spring Boot* se procede a crear un nuevo proyecto que se denomine *apimonedas* y que contenga las siguientes dependencias:

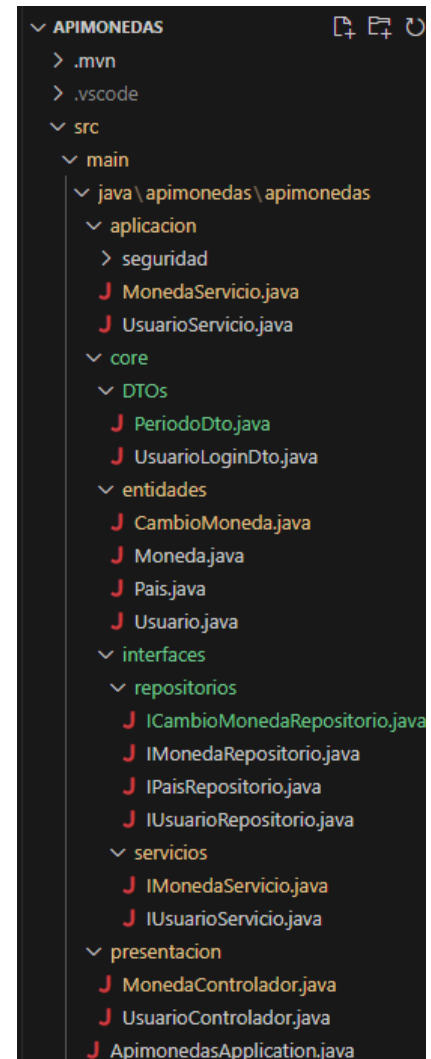
Grupo	Artefacto
org.springframework.boot	spring-boot-starter-web
	spring-boot-starter-data-jpa
	spring-boot-devtools
org.postgresql	postgresql

Acorde con las arquitecturas limpias, se plantea la siguiente distribución de las clases que componen el proyecto, acorde con la arquitectura *Onion* ya mencionada:

- Paquete *apimonedas.apimonedas.core.entidades* con las clases correspondientes a las **Entidades**, las cuales representan objetos o conceptos del mundo real que se almacenan y se gestionan en una base de datos. Es la capa más independiente de todas.

Para que *Spring Boot* y JPA reconozcan una clase como una entidad, esta debe estar anotada con la anotación **@Entity**.

Las entidades juegan un papel importante en el mapeo objeto-relacional (ORM) y permiten interactuar con la base de datos de manera orientada a objetos, lo que facilita la persistencia y recuperación de datos en las aplicaciones *Spring Boot*.



- Paquete *apimonedas.apimonedas.core.repositorios* con las clases correspondientes a los **Repositorios**, los cuales son interfaces que se utilizan para interactuar con una base de datos y realizar operaciones de persistencia de datos. Los repositorios proporcionan una abstracción sobre las operaciones de acceso a la base de datos, lo que simplifica la forma en que se interactúa con los datos almacenados en una base de datos relacional. La gestión de la base de datos se realiza generalmente utilizando tecnologías de persistencia como JPA (*Java Persistence API*).

Para que *Spring Boot* reconozca una interfaz o clase como un repositorio, generalmente se anota con la anotación **@Repository**. Esta anotación informa a *Spring* que la interfaz es un componente de acceso a datos y debe ser gestionada por el contenedor de *Spring*.

Una **Interfaz** es una definición de un contrato o conjunto de métodos que una clase debe implementar. Las interfaces en *Spring Boot* se utilizan para definir contratos que pueden ser implementados por clases concretas. Son una parte fundamental

de la programación orientada a objetos y se utilizan para establecer un conjunto de métodos que deben estar disponibles en las clases que las implementen.

Las interfaces proporcionan una abstracción que permite que las clases concretas proporcionen una implementación específica para los métodos definidos en la interfaz. Esto promueve la modularidad y el desacoplamiento en la programación

Los repositorios definen métodos para realizar operaciones de lectura (consultas) y escritura (inserciones, actualizaciones, eliminaciones) en la base de datos. Estos métodos siguen una convención de nomenclatura basada en la firma del método y la convención de nombres, lo que permite a *Spring Boot* generar consultas SQL automáticamente a partir de los nombres de los métodos.

Los repositorios suelen trabajar en conjunto con JPA para proporcionar una capa de abstracción sobre la base de datos. *Spring Boot* maneja la creación y administración de instancias de repositorios, lo que simplifica la configuración y el uso. Lo anterior significa que no hay que realizar implementaciones de tales interfaces.

- Paquete *apimonedas.apimonedas.core.servicios* con los archivos correspondientes a las interfaces propias de los servicios.

En este sentido y para garantizar el desacoplamiento de las capas superiores (como la de los **Controladores**) con respecto a las inferiores (como la de **Repositorios**), la funcionalidad de los servicios será definida en interfaces cuyas implementaciones irán en otro paquete.

- Paquete *apimonedas.apimonedas.aplicacion* con las clases correspondientes a los servicios.

Un **servicio** es una componente de la aplicación que encapsula la lógica de negocio y proporciona funcionalidades específicas a otras partes de la aplicación, como controladores, otros servicios o componentes. Los servicios son una parte importante del patrón de arquitectura de software conocido como "Inversión de Control" o "Inyección de Dependencias". En *Spring Boot*, los servicios se crean generalmente como clases Java anotadas con **@Service** para que Spring los administre y los haga disponibles para su uso en la aplicación.

Spring Boot utiliza la **Inyección de Dependencias** para proporcionar instancias de servicios a otras partes de la aplicación que los necesitan. Esto significa que no se necesitan crear manualmente instancias de servicios; Spring se encarga de ello.

La inyección de dependencias es un patrón de diseño y un principio de programación utilizado en el desarrollo de software para administrar y proporcionar las dependencias que un componente o clase necesita para funcionar. En lugar de que una clase cree o instancie sus propias dependencias, la inyección de dependencias permite que las dependencias sean proporcionadas desde el exterior, generalmente a través de la configuración de la aplicación o un contenedor de inversión de control (IoC) como Spring en el contexto de *Spring Boot*. Esto promueve el modularidad, el desacoplamiento y la facilidad de prueba al permitir que las

dependencias se intercambien o modifiquen fácilmente sin cambiar la clase que las utiliza.

Un ejemplo de como se aplica este patrón, se puede observar en las primeras instrucciones de la clase que implementa el servicio para Campeonatos:

```
@Service
public class MonedaServicio implements IMonedaServicio {

    private IMonedaRepositorio repositorio;
    private ICambioMonedaRepositorio repositorioCambio;

    public MonedaServicio(IMonedaRepositorio repositorio,
        ICambioMonedaRepositorio repositorioCambio) {
        this.repositorio = repositorio;
        this.repositorioCambio = repositorioCambio;
    }
    ...
}
```

Aquí en el método constructor de la clase se inyecta los objetos de los repositorio (necesario para realizar las consultas a la base de datos) el cual se asigna a un atributo privado. Esto significa que la instancia viene desde el exterior de la clase (y realmente quien la gestiona es *Spring Boot*).

Spring Boot también permite la inyección de las dependencias a través de anotaciones como **@Autowired**

```
@Service
public class UsuarioServicio implements IUsuarioServicio {

    @Autowired
    private IUsuarioRepositorio repositorio;

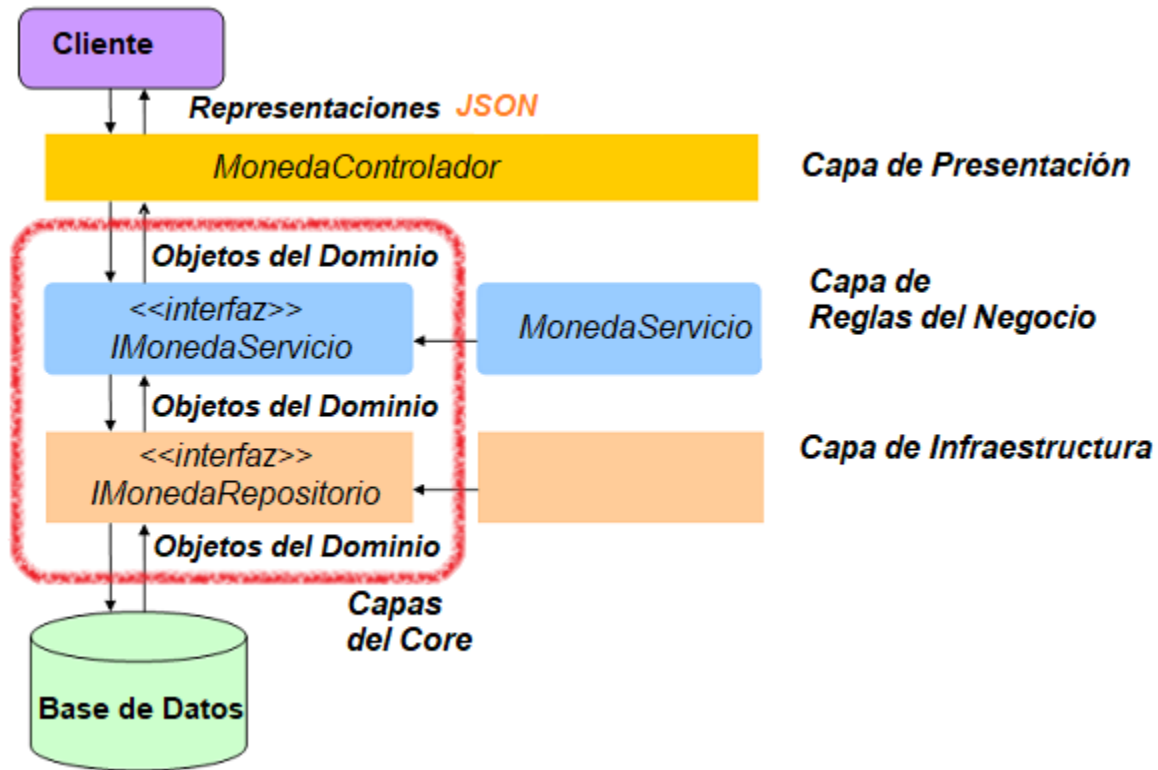
    @Autowired
    private SeguridadServicio servicioSeguridad;
    ...
}
```

- Paquete *apimonedas.apimonedas.presentacion* con las clases correspondientes a los controladores.

Un **Controlador** (*Controller* en inglés) es una clase que forma parte de la capa de presentación de una aplicación web. Se utilizan para manejar las solicitudes HTTP entrantes, procesarlas y generar respuestas apropiadas que se envían de vuelta al cliente, que puede ser un navegador web, una aplicación móvil u otro sistema. Los controladores en *Spring Boot* se suelen anotar con **@Controller** para indicar que son componentes de controlador gestionados por *Spring*.

Los métodos dentro de un controlador, también conocidos como "métodos de manejo de solicitudes" o "métodos web", se anotan con **@RequestMapping**

El siguiente diagrama ilustra arquitectónicamente como se distribuyen las clases por ejemplo para la funcionalidad asociada a *Monedas*:



Mapeando las entidades

De acuerdo al modelo relacional y el requerimiento, el siguiente es el mapeado de clases necesarias.

Tener en cuenta que:

- ✓ Toda entidad debe tener la anotación **@Entity**
- ✓ Se debe agregar también la anotación **@Table** en la cual se especifica mediante el atributo **name** el nombre de la tabla asociada
- ✓ Cada campo debe tener la anotación **@Column** la cual especifica el nombre del campo (atributo **name**), la longitud en caso de ser String (atributo **length**), exigir valores únicos (atributo **unique**), posibilidad de tener valores nulos (atributo **nullable**)
- ✓ El campo clave primaria se indica mediante la anotación **@Id**
- ✓ Como la mayoría de los campos de las claves primarias están definidos como autonuméricos, se deben utilizar las anotaciones **@GeneratedValue** y **@GenericGenerator** para especificar la estrategia de autonumerado.
- ✓ Las claves foráneas se indican mediante la anotación **@JoinColumn** donde se especifica el nombre del campo (atributo **name**) y el nombre del campo en la tabla referenciada (atributo **referencedColumnName**). Es importante anotar que el tipo asociado a la variable debe ser otra entidad (que corresponde a la tabla relacionada)
- ✓ Sobre las claves foráneas también debe ir la anotación **@ManyToOne** la cual indica la cardinalidad entre las entidades relacionadas
- ✓ Las clases deben incluir los métodos constructores, los getters y los setters.

- Clase *Moneda*

```
package apimonedas.apimonedas.core.entidades;

import org.hibernate.annotations.GenericGenerator;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name="moneda")
public class Moneda {

    @Id
    @Column(name = "id")
    @GeneratedValue(strategy = GenerationType.AUTO, generator =
"secuencia_moneda")
    @GenericGenerator(name = "secuencia_moneda", strategy =
"increment")
    private long id;

    @Column(name = "moneda", length = 100, unique = true)
    private String nombre;

    @Column(name = "sigla", length = 5, unique = true)
    private String sigla;

    @Column(name = "simbolo", length = 5)
    private String simbolo;

    @Column(name = "emisor", length = 100)
    private String emisor;

    public Moneda() {
    }

    public Moneda(long id, String nombre, String sigla, String
simbolo, String emisor) {
        this.id = id;
        this.nombre = nombre;
        this.sigla = sigla;
        this.simbolo = simbolo;
        this.emisor = emisor;
    }

    public long getId() {
        return id;
    }

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

```
}

public String getNombre() {
    return nombre;
}

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

public String getSigla() {
    return sigla;
}

public void setSigla(String sigla) {
    this.sigla = sigla;
}

public String getSimbolo() {
    return simbolo;
}

public void setSimbolo(String simbolo) {
    this.simbolo = simbolo;
}

public String getEmisor() {
    return emisor;
}

public void setEmisor(String emisor) {
    this.emisor = emisor;
}
}
```

En esta clase en particular se agregó una variable tipo *List* compuesta por entidades *Grupo* para indicar una relación uno a muchos (anotación **@OneToMany**) entre las entidades *Campeonato* y *Grupo*. También se debe colocar la anotación **@JsonIgnore** para indicar que esta variable no debe incluirse en la representación JSON del objeto, es decir, que debe ser ignorado durante la serialización. Para ello se tiene un método *getter*.

- **Clase Pais**

```
package apimonedas.apimonedas.core.entidades;

import org.hibernate.annotations.GenericGenerator;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
```

```
import jakarta.persistence.Table;

@Entity
@Table(name = "pais")
public class Pais {

    @Id
    @Column(name = "id")
    @GeneratedValue(strategy = GenerationType.AUTO, generator =
"secuencia_pais")
    @GenericGenerator(name = "secuencia_pais", strategy =
"increment")
    private long id;

    @Column(name = "pais", length = 100, unique = true)
    private String nombre;

    @Column(name = "codigoalfa2")
    private String codigoAlfa2;

    @Column(name = "codigoalfa3")
    private String codigoAlfa3;

    @ManyToOne
    @JoinColumn(name = "idmoneda", referencedColumnName = "id")
    private Moneda moneda;

    public Pais() {
    }

    public Pais(long id, String nombre, String codigoAlfa2, String
codigoAlfa3, Moneda moneda) {
        this.id = id;
        this.nombre = nombre;
        this.codigoAlfa2 = codigoAlfa2;
        this.codigoAlfa3 = codigoAlfa3;
        this.moneda = moneda;
    }

    public long getId() {
        return id;
    }

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

    public String getNombre() {
        return nombre;
    }

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

```
public String getCodigoAlfa2() {
    return codigoAlfa2;
}

public void setCodigoAlfa2(String codigoAlfa2) {
    this.codigoAlfa2 = codigoAlfa2;
}

public String getCodigoAlfa3() {
    return codigoAlfa3;
}

public void setCodigoAlfa3(String codigoAlfa3) {
    this.codigoAlfa3 = codigoAlfa3;
}

public Moneda getMoneda() {
    return moneda;
}

public void setMoneda(Moneda moneda) {
    this.moneda = moneda;
}
}
```

- **Clase *CambioMoneda***

```
package apimonedas.apimonedas.core.entidades;

import java.sql.Date;

import org.hibernate.annotations.GenericGenerator;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.Table;

@Entity
@Table(name = "cambiomoneda")
public class CambioMoneda {

    @Id
    @Column(name = "id")
    @GeneratedValue(strategy = GenerationType.AUTO, generator =
"CambioMoneda_Secuencia")
    @GenericGenerator(name = "CambioMoneda_Secuencia", strategy =
"increment")
    private long id;
```

```
@ManyToOne
@JoinColumn(name = "idmoneda", referencedColumnName = "id")
private Moneda moneda;

@Column(name = "fecha")
private Date fecha;

@Column(name = "cambio")
private double valor;

public CambioMoneda() {
}

public CambioMoneda(Moneda moneda, Date fecha, double valor) {
    this.moneda = moneda;
    this.fecha = fecha;
    this.valor = valor;
}

public Moneda getMoneda() {
    return moneda;
}

public void setMoneda(Moneda moneda) {
    this.moneda = moneda;
}

public Date getFecha() {
    return fecha;
}

public void setFecha(Date fecha) {
    this.fecha = fecha;
}

public double getValor() {
    return valor;
}

public void setValor(double valor) {
    this.valor = valor;
}
}
```

- Clase *Usuario* (Clase que originalmente no está en el modelo relacional pero se requiere para validar las credenciales válidas de acceso a la API)

```
package apimonedas.apimonedas.core.entidades;

import org.hibernate.annotations.GenericGenerator;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
```



```
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name = "usuario")
public class Usuario {

    @Id
    @Column(name = "id")
    @GeneratedValue(strategy = GenerationType.AUTO, generator =
"secuencia_usuario")
    @GenericGenerator(name = "secuencia_usuario", strategy =
"increment")
    private long id;

    @Column(name = "usuario", length = 100, unique = true)
    private String usuario;

    @Column(name = "nombre", length = 100)
    private String nombre;

    @Column(name = "clave")
    private String clave;

    @Column(name = "roles")
    private String roles;

    public String getRoles() {
        return roles;
    }

    public void setRoles(String roles) {
        this.roles = roles;
    }

    public Usuario(long id, String usuario, String nombre) {
        this.id = id;
        this.usuario = usuario;
        this.nombre = nombre;
    }

    public Usuario() {
    }

    public long getId() {
        return id;
    }

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

    public String getUsuario() {
        return usuario;
    }
}
```

```
public void setUsuario(String usuario) {
    this.usuario = usuario;
}

public String getNombre() {
    return nombre;
}

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

public String getClave() {
    return clave;
}
}
```

Repositorios

Para este ejercicio se tendrán los siguientes repositorios.

- Interfaz *MonedaRepositorio* que define los métodos para operar con la entidad *Moneda*

```
package apimonedas.apimonedas.core.interfaces.repositorios;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;

import java.util.List;

import apimonedas.apimonedas.core.entidades.Moneda;

@Repository
public interface IMonedaRepositorio extends JpaRepository<Moneda, Long> {

    @Query("SELECT m FROM Moneda m WHERE m.nombre LIKE '%' || ?1 || '%'"')
    List<Moneda> buscar(String nombre);

    @Query("SELECT m FROM Pais p JOIN p.moneda m WHERE p.nombre=?1")
    Moneda buscarPorPais(String nombre);

}
```

- Interfaz *CambioMonedaRepositorio* que define los métodos para operar con la entidad *CambioMoneda*

```
package apimonedas.apimonedas.core.interfaces.repositorios;

import java.util.Date;
import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
```

```
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;

import apimonedas.apimonedas.core.entidades.CambioMoneda;

@Repository
public interface ICambioMonedaRepositorio extends
JpaRepository<CambioMoneda, Long> {

    @Query("SELECT cm FROM CambioMoneda cm WHERE cm.moneda.id=?1 AND
cm.fecha >= ?2 AND cm.fecha <= ?3")
    public List<CambioMoneda> listarPorPeriodo(long idMoneda, Date
fecha1, Date fecha2);

}
```

- Interfaz *UsuarioRepositorio* que define los métodos para operar con la entidad *Usuario*

```
package apimonedas.apimonedas.core.interfaces.repositorios;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;

import apimonedas.apimonedas.core.entidades.Usuario;

@Repository
public interface IUsuarioRepositorio extends JpaRepository<Usuario,
Long>{

    @Query("SELECT u FROM Usuario u WHERE u.usuario = ?1")
    Usuario obtener(String usuario);

    @Query("SELECT u FROM Usuario u WHERE u.usuario=?1 AND
u.clave=?2")
    Usuario validarUsuario(String usuario, String clave);

}
```

Contratos para los Servicios

Se definen los siguientes contratos para los servicios:

- Interfaz *IMonedaServicio* que incluye los métodos para buscar la moneda de un país y listar los cambios de moneda durante un período:

```
package apimonedas.apimonedas.core.interfaces.servicios;

import java.util.Date;
import java.util.List;

import apimonedas.apimonedas.core.entidades.CambioMoneda;
import apimonedas.apimonedas.core.entidades.Moneda;

public interface IMonedaServicio {
```

```
public List<Moneda> listar();

public Moneda obtener(Long id);

public List<Moneda> buscar(String nombre);

public Moneda buscarPorPais(String nombre);

public Moneda agregar(Moneda moneda);

public Moneda modificar(Moneda moneda);

public boolean eliminar(Long id);

public List<CambioMoneda> listarPorPeriodo(long idMoneda, Date
fecha1, Date fecha2);
}
```

- Interfaz *IUsuarioServicio* que incluye el método para validar si a un usuario se le concede acceso:

```
package apimonedas.apimonedas.core.interfaces.servicios;

import java.util.List;

import apimonedas.apimonedas.core.DTOs.UsuarioLoginDto;
import apimonedas.apimonedas.core.entidades.Usuario;

public interface IUsuarioServicio {

    public UsuarioLoginDto login(String nombreUsuario, String
clave);

    public List<Usuario> listar();

    public Usuario obtener(Long id);

    public List<Usuario> buscar(String nombre);

    public Usuario agregar(Usuario Usuario);

    public Usuario modificar(Usuario Usuario);

    public boolean eliminar(Long id);
}
```

Servicios

Ahora bien, se implementan los anteriores contratos mediante las siguientes clases:

- Clase *MonedaServicio*

```
package apimonedas.apimonedas.aplicacion;
```

```
import java.util.Date;
import java.util.List;
import java.util.Optional;

import org.springframework.stereotype.Service;

import apimonedas.apimonedas.core.entidades.CambioMoneda;
import apimonedas.apimonedas.core.entidades.Moneda;
import
apimonedas.apimonedas.core.interfaces.repositorios.ICambioMonedaRep
ositorio;
import
apimonedas.apimonedas.core.interfaces.repositorios.IMonedaRepositor
io;
import
apimonedas.apimonedas.core.interfaces.servicios.IMonedaServicio;

@Service
public class MonedaServicio implements IMonedaServicio {

    private IMonedaRepositorio repositorio;
    private ICambioMonedaRepositorio repositorioCambio;

    public MonedaServicio(IMonedaRepositorio repositorio,
        ICambioMonedaRepositorio repositorioCambio) {
        this.repositorio = repositorio;
        this.repositorioCambio = repositorioCambio;
    }

    @Override
    public List<Moneda> listar() {
        return repositorio.findAll();
    }

    @Override
    public Moneda obtener(Long id) {
        var moneda = repositorio.findById(id);
        return moneda.isEmpty() ? null : moneda.get();
    }

    @Override
    public List<Moneda> buscar(String nombre) {
        return repositorio.buscar(nombre);
    }

    @Override
    public Moneda buscarPorPais(String nombre) {
        return repositorio.buscarPorPais(nombre);
    }

    public Moneda agregar(Moneda moneda) {
        moneda.setId(0);
        return repositorio.save(moneda);
    }
}
```

```
@Override
public Moneda modificar(Moneda moneda) {
    Optional<Moneda> monedaEncontrado =
repositorio.findById(moneda.getId());
    if (!monedaEncontrado.isEmpty()) {
        return repositorio.save(moneda);
    } else {
        return null;
    }
}

@Override
public boolean eliminar(Long id) {
    try {
        repositorio.deleteById(id);
        return true;
    } catch (Exception ex) {
        return false;
    }
}

@Override
public List<CambioMoneda> listarPorPeriodo(long idMoneda, Date
fecha1, Date fecha2) {
    return repositorioCambio.listarPorPeriodo(idMoneda, fecha1,
fecha2);
}
}
```

Aquí es importante reconocer los métodos que se heredan de la interfaz *JpaRepository*:

Método	Descripción
<code>findAll()</code>	Permite listar todos los registros
<code>findById()</code>	Obtiene el registro correspondiente a una clave primaria
<code>save()</code>	Guarda los datos de un registro
<code>deleteById()</code>	Elimina el registro identificado con una clave primaria

Por otro lado, es importante tener en cuenta que el método del repositorio *findById()* retorna un objeto de tipo *Optional* el cual sirve para representar un valor que puede estar opcionalmente presente o ausente. Es una forma de evitar problemas de *NullPointerException* al trabajar con valores que pueden ser nulos.

En este caso, cuando se trata de listar los grupos de un campeonato, se valida que la clave primaria proporcionada si corresponda a un campeonato existente mediante la instrucción *isPresent()*.

- **Clase *UsuarioServicio*:**

```
package apimonedas.apimonedas.aplicacion;

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

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import
apimonedas.apimonedas.aplicacion.seguridad.SeguridadServicio;
import apimonedas.apimonedas.core.DTOs.UsuarioLoginDto;
import apimonedas.apimonedas.core.entidades.Usuario;
import
apimonedas.apimonedas.core.interfaces.repositorios.IUsuarioReposito
rio;
import
apimonedas.apimonedas.core.interfaces.servicios.IUsuarioServicio;

@Service
public class UsuarioServicio implements IUsuarioServicio {

    @Autowired
    private IUsuarioRepositorio repositorio;

    @Autowired
    private SeguridadServicio servicioSeguridad;

    @Override
    public UsuarioLoginDto login(String nombreUsuario, String clave)
    {
        Usuario usuarioObtenido =
repositorio.validarUsuario(nombreUsuario, clave);
        UsuarioLoginDto userLoginResponseDto = new
UsuarioLoginDto(usuarioObtenido);
        if (usuarioObtenido != null) {

userLoginResponseDto.setToken(servicioSeguridad.generarToken(nombre
Usuario));
        }
        return userLoginResponseDto;
    }

    @Override
    public List<Usuario> listar() {
        return repositorio.findAll();
    }

    @Override
    public Usuario obtener(Long id) {
        var usuario = repositorio.findById(id);
        return usuario.isEmpty() ? null : usuario.get();
    }

    @Override
    public List<Usuario> buscar(String nombre) {
        return repositorio.buscar(nombre);
    }

    public Usuario agregar(Usuario usuario) {
        usuario.setId(0);
    }
}
```

```
        return repositorio.save(usuario);
    }

    @Override
    public Usuario modificar(Usuario usuario) {
        Optional<Usuario> usuarioEncontrado =
repositorio.findById(usuario.getId());
        if (!usuarioEncontrado.isEmpty()) {
            return repositorio.save(usuario);
        } else {
            return null;
        }
    }

    @Override
    public boolean eliminar(Long id) {
        try {
            repositorio.deleteById(id);
            return true;
        } catch (Exception ex) {
            return false;
        }
    }
}
```

En este caso si un usuario envía las credenciales validas, se hace uso de la clase *SeguridadServicio* para generar un token válido. Más adelante se expondrá el manejo de la seguridad mediante tokens.

Controladores

Las siguientes clases tendrán la anotación *@RestController* indicando que gestionan solicitudes HTTP y devuelven respuestas en formato de datos, generalmente en formato JSON, adecuado para aplicaciones web *RESTfull*.

Se debe observar que la ruta del método la componen las anotaciones *@RequestMapping* de la clase y la del método. Esta iría después de la ruta del servidor. Esta anotación también permite especificar además de la ruta, el tipo de método que en todos los casos de este ejercicio será *RequestMethod.GET*.

- Clase *MonedaControlador* con los métodos web para para buscar la moneda de un país y listar los cambios de moneda durante un período:

```
package apimonedas.apimonedas.presentacion;

import java.util.List;

import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
```



```
import apimonedas.apimonedas.core.DTOs.PeriodoDto;
import apimonedas.apimonedas.core.entidades.CambioMoneda;
import apimonedas.apimonedas.core.entidades.Moneda;
import
apimonedas.apimonedas.core.interfaces.servicios.IMonedaServicio;

@RestController
@RequestMapping("/api/monedas")
public class MonedaControlador {
    private IMonedaServicio servicio;

    public MonedaControlador(IMonedaServicio servicio) {
        this.servicio = servicio;
    }

    @RequestMapping(value = "/listar", method = RequestMethod.GET)
    public List<Moneda> listar() {
        return servicio.listar();
    }

    @RequestMapping(value = "/obtener/{id}", method =
RequestMethod.GET)
    public Moneda obtener(@PathVariable long id) {
        return servicio.obtener(id);
    }

    @RequestMapping(value = "/buscar/{nombre}", method =
RequestMethod.GET)
    public List<Moneda> buscar(@PathVariable String nombre) {
        return servicio.buscar(nombre);
    }

    @RequestMapping(value = "/buscarporpais/{nombre}", method =
RequestMethod.GET)
    public Moneda buscarPorPais(@PathVariable String nombre) {
        return servicio.buscarPorPais(nombre);
    }

    @RequestMapping(value = "/agregar", method = RequestMethod.POST)
    public Moneda crear(@RequestBody Moneda moneda) {
        return servicio.agregar(moneda);
    }

    @RequestMapping(value = "/modificar", method =
RequestMethod.PUT)
    public Moneda actualizar(@RequestBody Moneda moneda) {
        return servicio.modificar(moneda);
    }

    @RequestMapping(value = "/eliminar/{id}", method =
RequestMethod.DELETE)
    public boolean eliminar(@PathVariable long id) {
        return servicio.eliminar(id);
    }
}
```

```
@RequestMapping(value = "/listarporperiodo", method =
RequestMethod.GET)
public List<CambioMoneda> listarPorPeriodo(@RequestBody
PeriodoDto periodo) {
    System.out.println("periodo="+periodo.getDesde()+"
"+periodo.getHasta()+" idmoneda="+periodo.getIdMoneda());

    return servicio.listarPorPeriodo(periodo.getIdMoneda(),
periodo.getDesde(), periodo.getHasta());
}
}
```

En el caso del método *buscarporpais ()* en la ruta se incluye una variable (la cual va entre { }). La variable debe ir como parámetro de entrada del método con la anotación *@PathVariable*

- Clase *UsuarioControlador* con el método web para validar las credenciales del usuario:

```
package apimonedas.apimonedas.presentacion;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

import apimonedas.apimonedas.core.DTOs.UsuarioLoginDto;
import apimonedas.apimonedas.core.entidades.Usuario;
import apimonedas.apimonedas.core.interfaces.servicios.IUsuarioServicio;

@RestController
@RequestMapping("/api/usuarios")
public class UsuarioControlador {

    @Autowired
    private IUsuarioServicio servicio;

    @RequestMapping(value = "/validar/{nombreUsuario}/{clave}",
method = RequestMethod.GET)
    public UsuarioLoginDto login(@PathVariable String nombreUsuario,
@PathVariable String clave) {
        return servicio.login(nombreUsuario, clave);
    }

    @RequestMapping(value = "/listar", method = RequestMethod.GET)
    public List<Usuario> listar() {
        return servicio.listar();
    }
}
```

```
}

@RequestMapping(value = "/obtener/{id}", method =
RequestMethod.GET)
public Usuario obtener(@PathVariable long id) {
    return servicio.obtener(id);
}

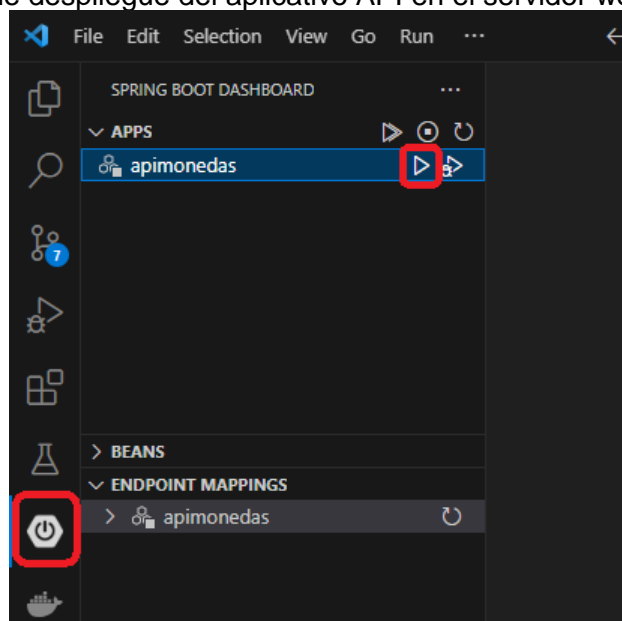
@RequestMapping(value = "/buscar/{nombre}", method =
RequestMethod.GET)
public List<Usuario> buscar(@PathVariable String nombre) {
    return servicio.buscar(nombre);
}

@RequestMapping(value = "/agregar", method = RequestMethod.POST)
public Usuario crear(@RequestBody Usuario usuario) {
    return servicio.agregar(usuario);
}

@RequestMapping(value = "/modificar", method =
RequestMethod.PUT)
public Usuario actualizar(@RequestBody Usuario usuario) {
    return servicio.modificar(usuario);
}

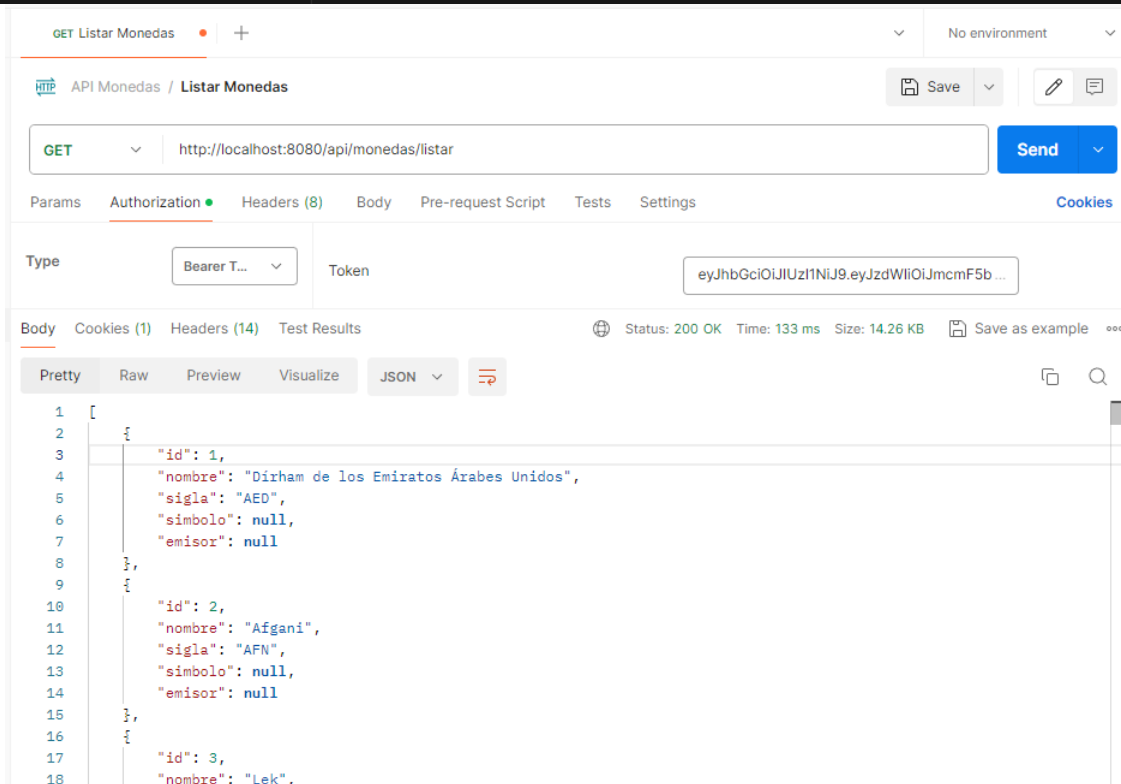
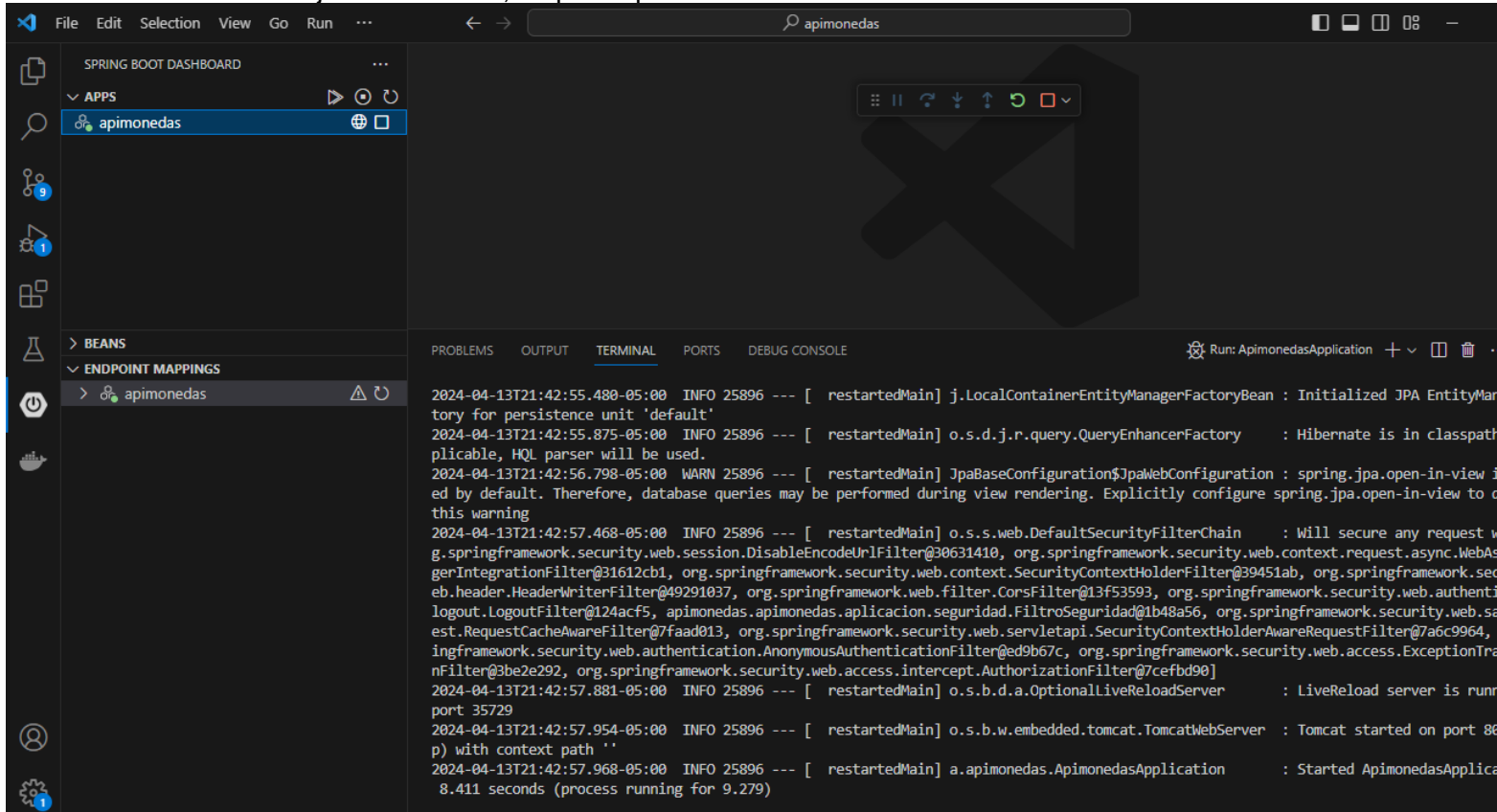
@RequestMapping(value = "/eliminar/{id}", method =
RequestMethod.DELETE)
public boolean eliminar(@PathVariable long id) {
    return servicio.eliminar(id);
}
}
```

Con esta codificación concluida, se dispone ya de un proyecto *Spring Boot* disponible para ser ejecutado. Para ello se accede al **Dashboard** de *Spring Boot*. El cual dispone de un botón de ejecución de despliegue del aplicativo API en el servidor web *Tomcat*:





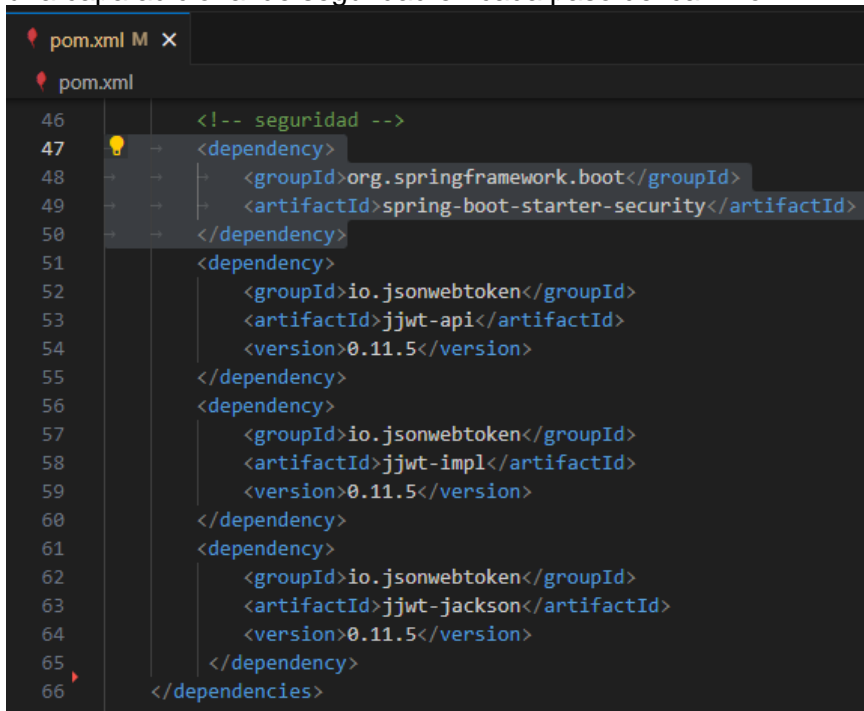
Estando en ejecución la API, se podrá probar su funcionamiento mediante *Postman*:



Seguridad basada en Tokens

La aplicación incluirá un sólido esquema de seguridad basado en tokens de portador (**Bearer Tokens**) para proteger los datos y garantizar un acceso seguro a la API. Con este enfoque, cada *request (solicitud)* debe incluir un token de autenticación válido en el *header (cabecera)* HTTP, lo que permite verificar la identidad del usuario y autorizar sus acciones de manera precisa y confiable.

Con *Spring Security* integrado en la aplicación, se asegura de que todas las interacciones estén protegidas mediante autenticación y autorización robustas. Además, el sistema de gestión de tokens garantiza la integridad y la confidencialidad de tus datos, proporcionando una capa adicional de seguridad en cada paso del camino.



```
46      <!-- seguridad -->
47      <dependency>
48          <groupId>org.springframework.boot</groupId>
49          <artifactId>spring-boot-starter-security</artifactId>
50      </dependency>
51      <dependency>
52          <groupId>io.jsonwebtoken</groupId>
53          <artifactId>jjwt-api</artifactId>
54          <version>0.11.5</version>
55      </dependency>
56      <dependency>
57          <groupId>io.jsonwebtoken</groupId>
58          <artifactId>jjwt-impl</artifactId>
59          <version>0.11.5</version>
60      </dependency>
61      <dependency>
62          <groupId>io.jsonwebtoken</groupId>
63          <artifactId>jjwt-jackson</artifactId>
64          <version>0.11.5</version>
65      </dependency>
66  </dependencies>
```

Concepto de token

Un **token** es una pieza de información que se utiliza para representar la autorización o la identidad de un usuario en un sistema informático. En el contexto de la seguridad informática, un token es comúnmente utilizado como una forma de autenticación y autorización.


Cuando un usuario se autentica en un sistema, generalmente se le proporciona un token que puede incluir información sobre sus derechos de acceso, roles, tiempo de expiración, entre otros detalles relevantes. Este token es luego enviado con cada solicitud al servidor para que el servidor pueda verificar la identidad y los privilegios del usuario sin necesidad de autenticarse nuevamente en cada solicitud.

Los tokens pueden tener diferentes formatos y mecanismos de generación, como tokens JWT (JSON Web Tokens), tokens de sesión, tokens de acceso OAuth, entre otros. Independientemente del formato, los tokens son una herramienta crucial en la




implementación de sistemas seguros, ya que permiten una autenticación eficiente y autorización de usuarios en aplicaciones web y servicios en línea.

Dive into passkeys, see MFA and other new features in action in this developer webinar →

 JWT

Debugger Libraries Introduction Ask

Crafted by  Auth0 by Okta

Algorithm HS256

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJmcmF5b3NvcmlvIiwiaWF0IjoxNzE3MDU3NDk4LCJleSI6MTYzMTMwNTkyOTh9.u5UHRZNc4uyaosiDc_51LUNUwW1Ts7dxdpRINKeCNBw0
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{  "alg": "HS256"}
```

PAYLOAD: DATA

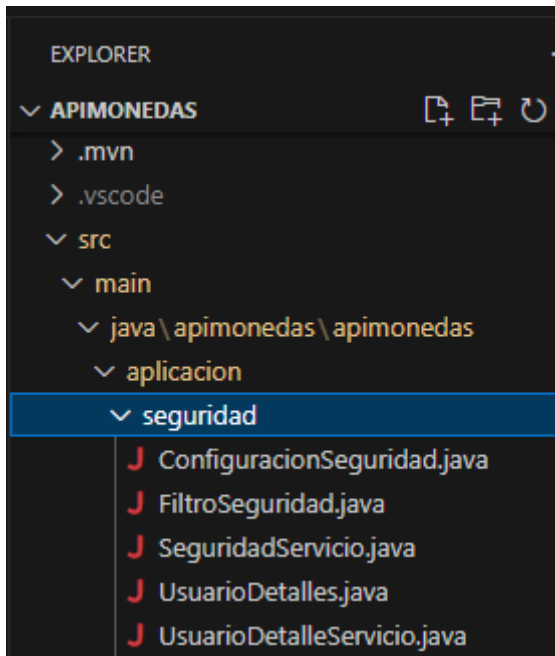
```
{  "sub": "frayosorio",  "iat": 1713057498,  "exp": 1713059298}
```

VERIFY SIGNATURE

```
HMACSHA256(  base64UrlEncode(header) + "." +  base64UrlEncode(payload),  your-256-bit-secret)
```

☐ secret base64 encoded

Las clases a codificar para poder aplicar seguridad a los métodos de la API serían las siguientes, las cuales se ubicarían den la capa de aplicación en un subpaquete que denominaremos *seguridad*:



En la **clase de configuración** de *Spring Security* se debe configurar cómo se realiza la autenticación y la autorización en la aplicación. Esto puede incluir la definición de rutas protegidas, la configuración del proveedor de autenticación (usando el *UserDetailsService* personalizado), la configuración de las reglas de autorización, entre otras configuraciones relacionadas con la seguridad.

La **clase de filtrado de solicitud** en *Spring Boot* se utiliza para interceptar las solicitudes HTTP que ingresan a la aplicación y aplicar lógica personalizada antes o después de que la solicitud alcance su destino final (por ejemplo, un controlador de *Spring MVC*). Una clase comúnmente utilizada para este propósito es *OncePerRequestFilter*, proporcionada por *Spring Framework*. Al crear una clase que extienda *OncePerRequestFilter* se garantiza que el método *doFilterInternal()* se ejecute solo una vez por solicitud, independientemente de cuántos filtros hayan sido aplicados.

En *Spring Security*, durante el **proceso de autenticación**, se necesita una manera de recuperar los detalles de un usuario, como su nombre de usuario, contraseña y roles, desde algún origen de datos (como una base de datos, un servicio web, LDAP, etc.). La interfaz *UserDetailsService* define un único método *loadUserByUsername()* que debe ser implementado. Este método recibe el nombre de usuario como argumento y devuelve un objeto que implementa la interfaz *UserDetails*.

El objeto *UserDetails* encapsula los detalles del usuario, como el nombre de usuario, la contraseña encriptada, los roles y otras propiedades relacionadas con la seguridad. Implementar la interfaz *UserDetailsService* permite a los desarrolladores personalizar cómo *Spring Security* carga los detalles de los usuarios desde el sistema de almacenamiento de usuarios de la aplicación.

Por lo tanto, al crear una implementación de *UserDetailsService*, se puede conectar *Spring Security* con el sistema propio de gestión de usuarios, permitiendo la autenticación y

autorización basadas en tus necesidades específicas de la aplicación. Esto proporciona una gran flexibilidad para integrar *Spring Security* con diferentes sistemas de gestión de usuarios.

- Clase *ConfiguracionSeguridad* con la definición de las rutas libres de token, las que lo requieren y el filtro aplicado:

```
package apimonedas.apimonedas.aplicacion.seguridad;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.method.configuration.EnableMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class ConfiguracionSeguridad {

    @Autowired
    private FiltroSeguridad filtro;

    @Bean
    public UserDetailsService servicioUsuario() {
        return new UsuarioDetalleServicio();
    }

    // Configurando Seguridad Http
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http
            .csrf(csrf -> csrf.disable()) // Deshabilita la protección CSRF
            .authorizeHttpRequests(
                (authz) -> authz
                    .requestMatchers("/api/usuarios/validar/**").permitAll()
                    // .requestMatchers("/api/monedas/**").permitAll()
                    .anyRequest().authenticated()
            )
        ;
    }
}
```



```
        )
        .addFilterAfter(filtro,
UsernamePasswordAuthenticationFilter.class)
        .build();

    }

}
```

- Clase *FiltroSeguridad* con la funcionalidad para procesar el *request* (Solicitud) en busca del token requerido. Incluye el filtrado de solicitudes HTTP.:

```
package apimonedas.apimonedas.aplicacion.seguridad;

import java.io.IOException;

import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.security.authentication.UsernamePasswordAuthent
icationToken;
import
org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import
org.springframework.security.web.authentication.WebAuthenticationDe
tailsSource;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import io.jsonwebtoken.ExpiredJwtException;
import io.jsonwebtoken.MalformedJwtException;
import io.jsonwebtoken.UnsupportedJwtException;
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

@Component
public class FiltroSeguridad extends OncePerRequestFilter {
    private final String HEADER = "Authorization";
    private final String PREFIX = "Bearer ";

    @Autowired
    private SeguridadServicio servicioSeguridad;

    @Autowired
    private UsuarioDetalleServicio servicioUsuario;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain)
        throws ServletException, IOException {
        try {
            String authHeader = request.getHeader(HEADER);
            String token = null;
            String nombreUsuario = null;
```

```
        if (authHeader != null && authHeader.startsWith(PREFIX))
        {
            token = authHeader.substring(7);
            nombreUsuario =
servicioSeguridad.extraerNombreUsuario(token);
        }
        if (nombreUsuario != null &&
SecurityContextHolder.getContext().getAuthentication() == null) {
            UserDetails userDetails =
servicioUsuario.loadUserByUsername(nombreUsuario);
            if (servicioSeguridad.validarToken(token,
userDetails)) {
                UsernamePasswordAuthenticationToken
autenticacionToken = new
UsernamePasswordAuthenticationToken(userDetails,
                null, userDetails.getAuthorities());
                autenticacionToken.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));

SecurityContextHolder.getContext().setAuthentication(autenticacionT
oken);
            }
        }
        filterChain.doFilter(request, response);
    } catch (ExpiredJwtException | UnsupportedJwtException |
MalformedJwtException e) {
        response.setStatus(HttpServletResponse.SC_FORBIDDEN);
        ((HttpServletResponse)
response).sendError(HttpServletResponse.SC_FORBIDDEN,
e.getMessage());
    }
}
}
```

- Clase *SeguridadServicio* con los métodos de generación y validación del token y el manejo de sus propiedades:

```
package apimonedas.apimonedas.aplicacion.seguridad;

import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import io.jsonwebtoken.io.Decoders;
import io.jsonwebtoken.security.Keys;
import io.jsonwebtoken.Claims;

import java.security.Key;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;

import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Component;
```



```
@Component
public class SeguridadServicio {
    public static final String SECRETO =
"5367566B59703373367639792F423F4528482B4D6251655468576D5A71347437";

    public String generarToken(String nombreUsuario) {
        Map<String, Object> declaraciones = new HashMap<>();
        return crearToken(declaraciones, nombreUsuario);
    }

    private String crearToken(Map<String, Object> declaraciones,
String nombreUsuario) {
        return Jwts.builder()
            .setClaims(declaraciones)
            .setSubject(nombreUsuario)
            .setIssuedAt(new Date(System.currentTimeMillis()))
            .setExpiration(new Date(System.currentTimeMillis() +
1000 * 60 * 30))
            .signWith(getClaveFirma(),
SignatureAlgorithm.HS256).compact();
    }

    private Key getClaveFirma() {
        byte[] keyBytes= Decoders.BASE64.decode(SECRETO);
        return Keys.hmacShaKeyFor(keyBytes);
    }

    public String extraerNombreUsuario(String token) {
        return extrearDeclaracion(token, Claims::getSubject);
    }

    public Date extrearExpiracion(String token) {
        return extrearDeclaracion(token, Claims::getExpiration);
    }

    public <T> T extrearDeclaracion(String token, Function<Claims,
T> declaracionesResolver) {
        final Claims declaraciones = extrearDeclaraciones(token);
        return declaracionesResolver.apply(declaraciones);
    }

    private Claims extrearDeclaraciones(String token) {
        return Jwts
            .parserBuilder()
            .setSigningKey(getClaveFirma())
            .build()
            .parseClaimsJws(token)
            .getBody();
    }

    private Boolean tokenExpirado(String token) {
        return extrearExpiracion(token).before(new Date());
    }
}
```

```
public Boolean validarToken(String token, UserDetails
userDetails) {
    final String nombreUsuario = extraerNombreUsuario(token);
    return (nombreUsuario.equals(userDetails.getUsername()) &&
!tokenExpirado(token));
}

}
```

- Clase *UsuarioDetalle* la cual representa a un usuario y su información asociada, como el nombre de usuario, la contraseña y los roles, utilizada para encapsular los detalles del usuario que *Spring Security* necesita durante la autenticación y la autorización.:

```
package apimonedas.apimonedas.aplicacion.seguridad;

import java.util.Arrays;
import java.util.Collection;
import java.util.List;
import java.util.stream.Collectors;

import org.springframework.security.core.GrantedAuthority;
import
org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

import apimonedas.apimonedas.core.entidades.Usuario;

public class UsuarioDetalles implements UserDetails {

    private String nombreUsuario;
    private String clave;
    private List<GrantedAuthority> permisos;

    public UsuarioDetalles(Usuario usuario) {
        nombreUsuario = usuario.getUsuario();
        clave = usuario.getClave();
        permisos = usuario.getRoles() != null ?
Arrays.stream(usuario.getRoles().split(","))
        .map(SimpleGrantedAuthority::new)
        .collect(Collectors.toList())
        : null;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return permisos;
    }

    @Override
    public String getPassword() {
        return clave;
    }

    @Override
```

```
public String getUsername() {
    return nombreUsuario;
}

@Override
public boolean isAccountNonExpired() {
    return true;
}

@Override
public boolean isAccountNonLocked() {
    return true;
}

@Override
public boolean isCredentialsNonExpired() {
    return true;
}

@Override
public boolean isEnabled() {
    return true;
}
}
```

- Clase *UsuarioDetalleService* que implementa la interfaz de *Spring Security* utilizada para recuperar los detalles de un usuario durante el proceso de autenticación:

```
package apimonedas.apimonedas.aplicacion.seguridad;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import apimonedas.apimonedas.core.entidades.Usuario;
import apimonedas.apimonedas.core.interfaces.repositorios.IUsuarioRepositorio;

@Service
public class UsuarioDetalleServicio implements UserDetailsService {

    @Autowired
    private IUsuarioRepositorio repositorio;

    @Override
    public UserDetails loadUserByUsername(String nombreUsuario)
        throws UsernameNotFoundException {
```



```
        Usuario          usuarioObtenido          =
repositorio.obtener(nombreUsuario);
        if (usuarioObtenido == null) {
            throw new UsernameNotFoundException(nombreUsuario);
        }
        return new UsuarioDetalles(usuarioObtenido);
    }
}
```