

Taller 6

Fecha: Abril de 2024

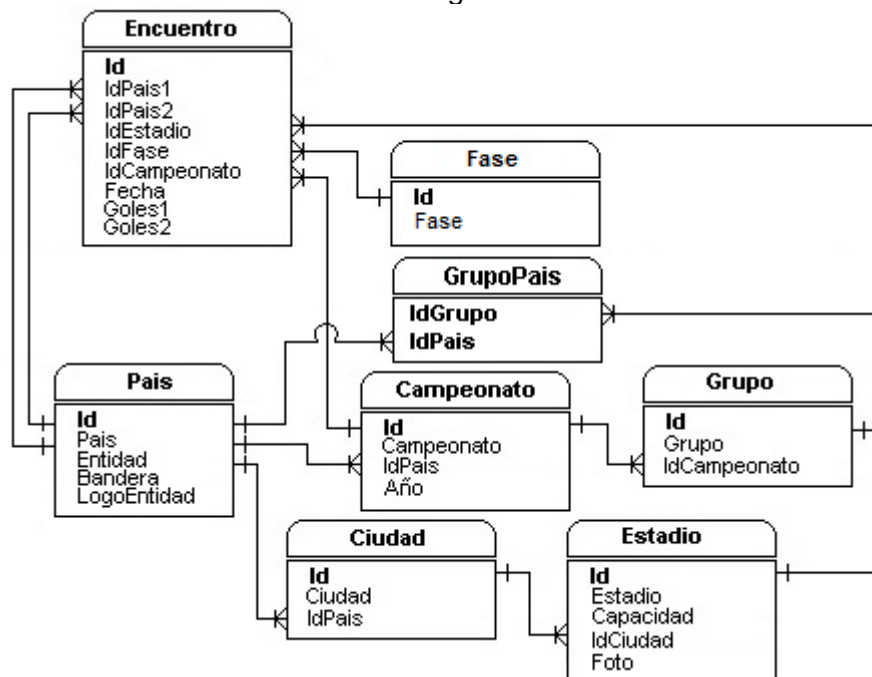
Indicador de logro a medir: Aplicar los conceptos de la lógica de programación, la orientación a objetos, el acceso y consulta de bases de datos relacionales en el desarrollo de una aplicación que provea funcionalidad a través de la web.

NOTAS:

- Este taller se debe hacer con carácter evaluativo (punto No 2). Representa en total una calificación de 15%.
- 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 campeonatos de fútbol de selecciones nacionales es el siguiente:



Se tiene una función de tabla que permite obtener la tabla de posiciones de un grupo determinado con base en los resultados de los encuentros.

Por ejemplo, si los resultados del grupo H del mundial Rusia 2018 fueron los siguientes:

	pais character varying (50)	goles1 integer	goles2 integer	pais character varying (50)	fecha date
1	Senegal	0	1	Colombia	2018-06-28
2	Polonia	0	3	Colombia	2018-06-24
3	Colombia	1	2	Japón	2018-06-19
4	Japón	0	1	Polonia	2018-06-28
5	Polonia	1	2	Senegal	2018-06-19
6	Japón	2	2	Senegal	2018-06-24

Se obtiene un resultado como el siguiente:

	id integer	pais character varying	pj integer	pg integer	pe integer	pp integer	gf integer	gc integer	puntos integer
1	1	Colombia	3	2	0	1	5	2	6
2	5	Japón	3	1	1	1	4	4	4
3	63	Senegal	3	1	1	1	4	4	4
4	62	Polonia	3	1	0	2	2	5	3

Se desea una API que permita realizar las siguientes operaciones:

- Listar los campeonatos FIFA que se encuentran registrados en la base de datos

GET
http://localhost:8080/campeonatos/listar
Send

Params
Authorization
Headers (6)
Body
Pre-request Script
Tests
Settings

Query Params

Body
Cookies
Headers (8)
Test Results

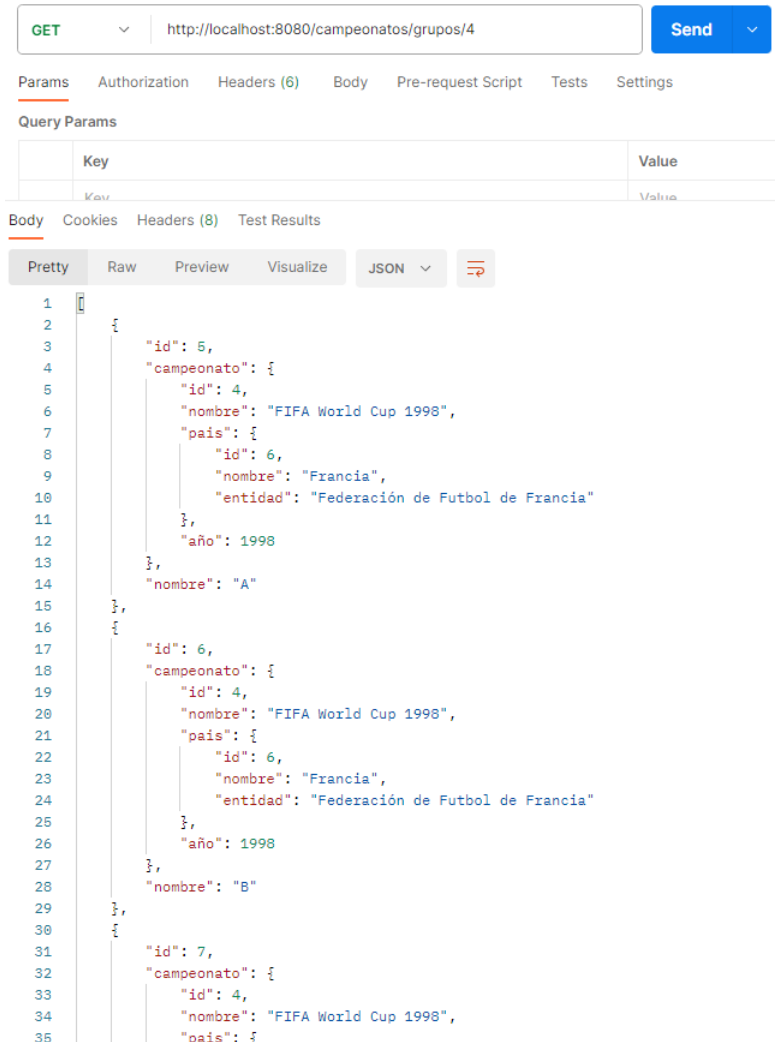
Pretty
Raw
Preview
Visualize
JSON

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
{
  "id": 1,
  "nombre": "FIFA World Cup 2010",
  "pais": {
    "id": 3,
    "nombre": "Sudáfrica",
    "entidad": "Sin Entidad"
  },
  "año": 2010
},
{
  "id": 2,
  "nombre": "FIFA World Cup 2006",
  "pais": {
    "id": 4,
    "nombre": "Alemania",
    "entidad": "Federación Alemana de Fútbol"
  },
  "año": 2006
},
{
  "id": 3,
  "nombre": "FIFA World Cup 2002",
  "pais": {
    "id": 5,
    "nombre": "Japón",
    "entidad": "Sin Entidad"
  },
  "año": 2002
},
{
  "id": 4,
  "nombre": "FIFA World Cup 1998",
  "pais": {

```

- Listar los grupos registrados para un campeonato en particular



GET http://localhost:8080/campeonatos/grupos/4 Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Query Params

Key	Value
Key	Value

Body Cookies Headers (8) Test Results

Pretty Raw Preview Visualize JSON

```

1  [
2    {
3      "id": 5,
4      "campeonato": {
5        "id": 4,
6        "nombre": "FIFA World Cup 1998",
7        "pais": {
8          "id": 6,
9          "nombre": "Francia",
10         "entidad": "Federación de Futbol de Francia"
11       },
12       "año": 1998
13     },
14     "nombre": "A"
15   },
16   {
17     "id": 6,
18     "campeonato": {
19       "id": 4,
20       "nombre": "FIFA World Cup 1998",
21       "pais": {
22         "id": 6,
23         "nombre": "Francia",
24         "entidad": "Federación de Futbol de Francia"
25       },
26       "año": 1998
27     },
28     "nombre": "B"
29   },
30   {
31     "id": 7,
32     "campeonato": {
33       "id": 4,
34       "nombre": "FIFA World Cup 1998",
35       "pais": {
  
```

- Dado un grupo, retornar la tabla de posiciones de las selecciones con base en los encuentros. El resultado de una solicitud en un cliente con *Postman* se puede observar en la siguiente gráfica:



GET ⌵ http://localhost:8080/grupos/5/posiciones Send ⌵

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
-----	-------	-------------	-----	-----------

Body Cookies Headers (5) Test Results ⌕ Status: 200 OK Time: 565 ms Size: 454 B Save Response ⌵

Pretty Raw Preview Visualize JSON ⌵ ⌵

```
1  
2  
3  "pais": "Brasil",  
4  "pJ": 3,  
5  "pG": 2,  
6  "pE": 0,  
7  "pP": 1,  
8  "gF": 6,  
9  "gC": 3,  
10 "puntos": 6  
11  
12  
13  "pais": "Noruega",  
14  "pJ": 3,  
15  "pG": 1,  
16  "pE": 2,  
17  "pP": 0,  
18  "gF": 5,  
19  "gC": 4,  
20 "puntos": 5  
21  
22  
23  "pais": "Marruecos",  
24  "pJ": 3,  
25  "pG": 1,  
26  "pE": 1,  
27  "pP": 1,  
28  "gF": 5,  
29  "gC": 5,  
30 "puntos": 4  
31  
32  
33  "pais": "Escocia",  
34  "pJ": 3,  
35  "pG": 0,  
36  "pE": 1,  
37  "pP": 2,  
38  "gF": 2,  
39  "gC": 6,  
40 "puntos": 1  
41  
42
```

R/

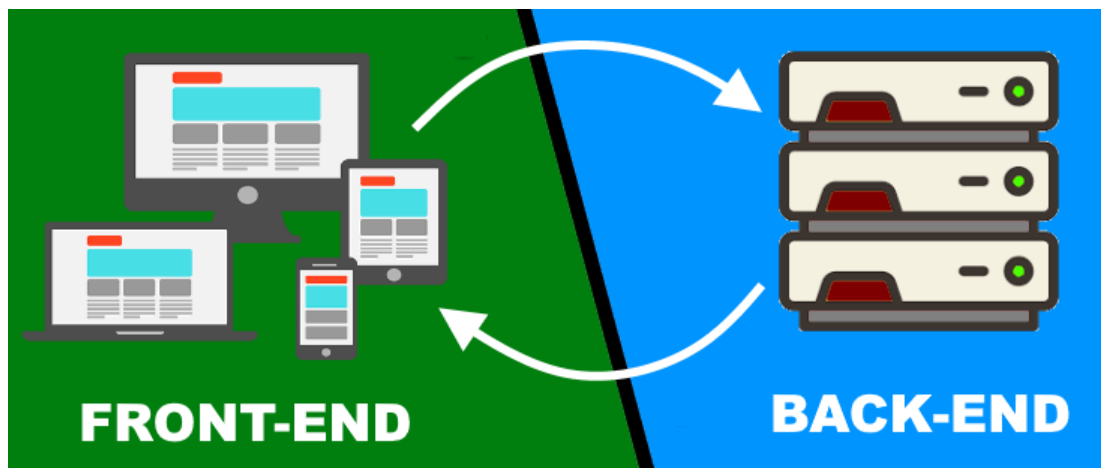
Para desarrollar este aplicativo web se tendrán las siguientes definiciones:

¿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</i> , <i>updates</i> , <i>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

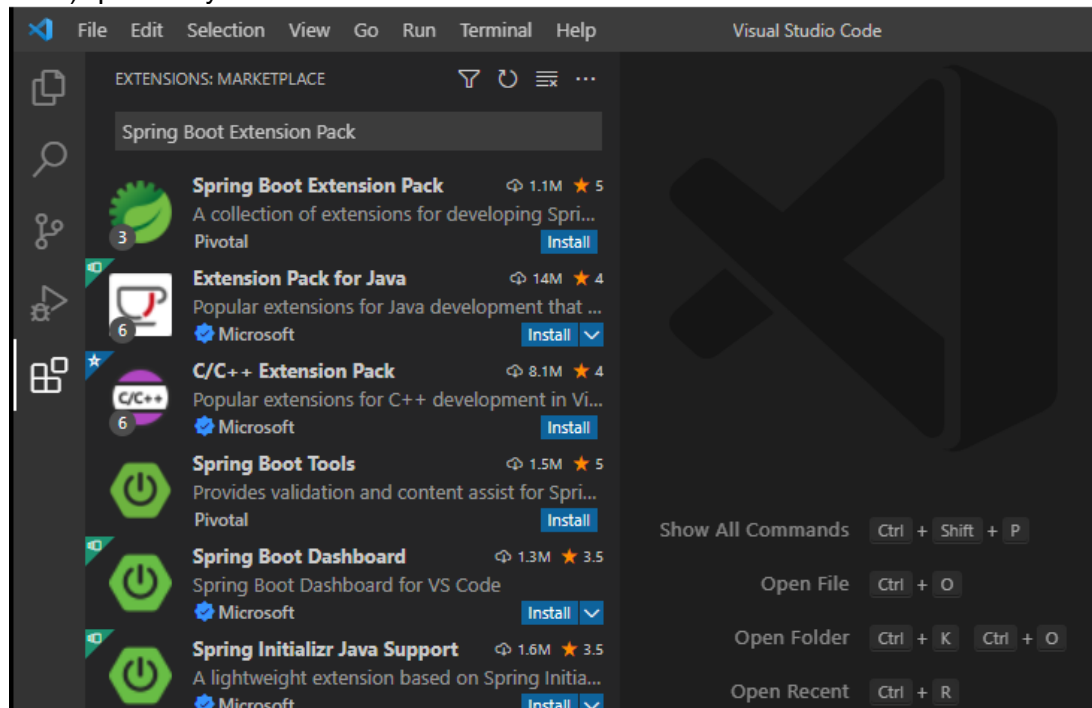
aplicación bajo la arquitectura que provee el framework de *Spring*, bien sea para interactuar con bases de datos, crear lógica de negocio o simplemente desarrollar un componente general para todas las capas de la aplicación.

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**.

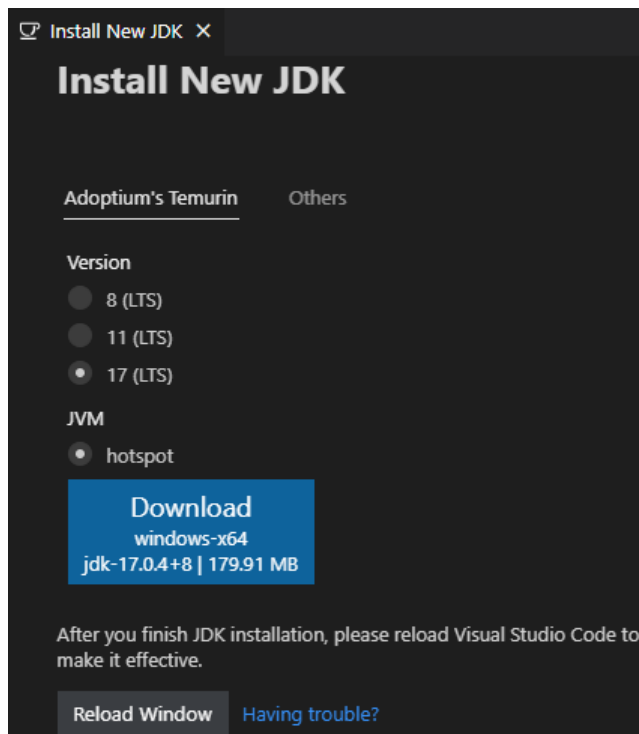
To verify it's installed, [create a new terminal](#) and try running the following command:

```
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
```

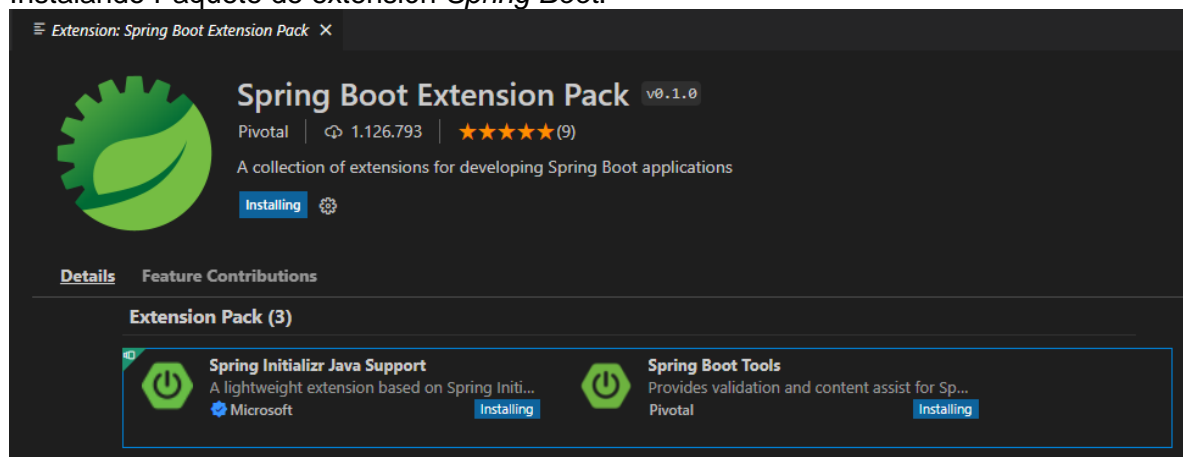
The detailed output will be based on the JDK you install.



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



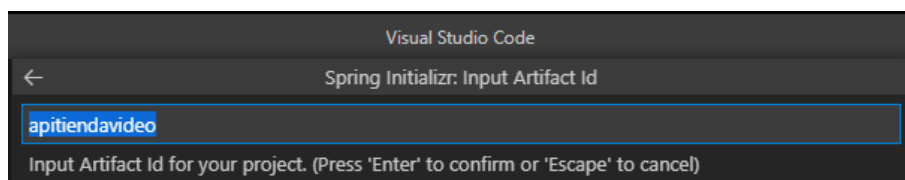
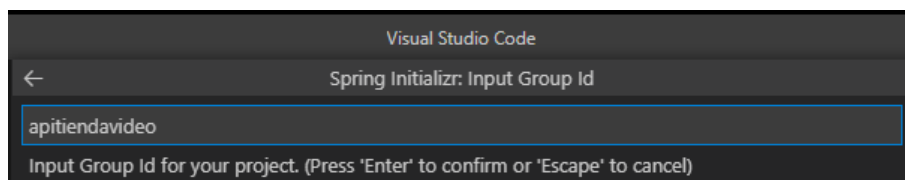
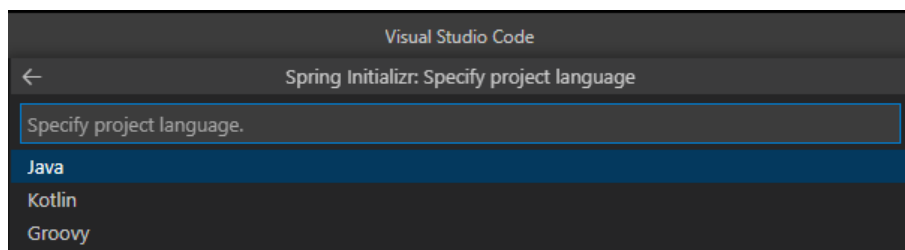
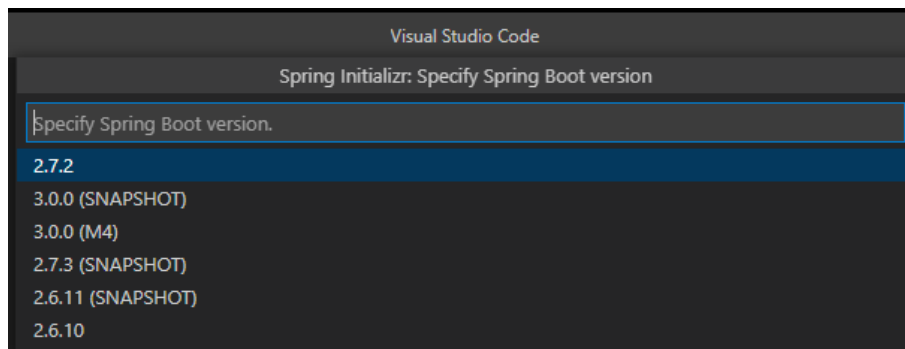
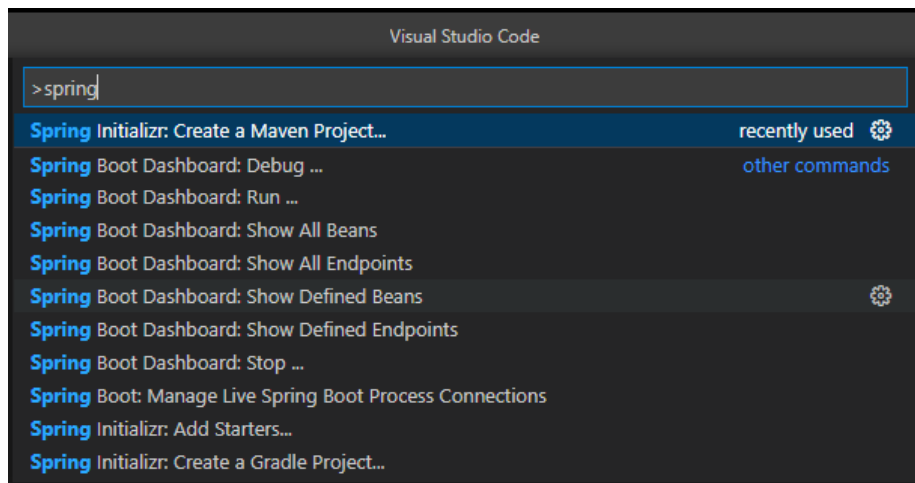
Instalando Paquete de extensión *Spring Boot*:

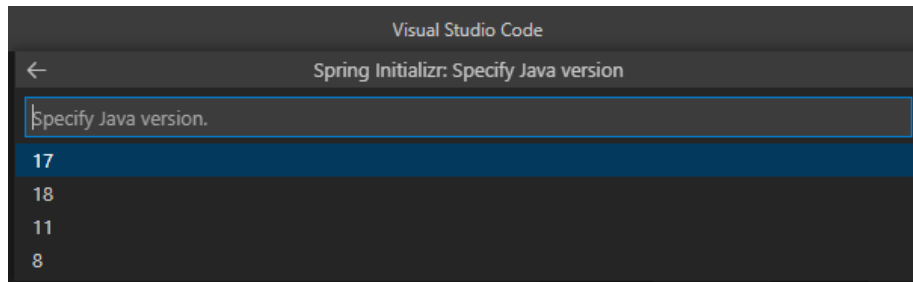
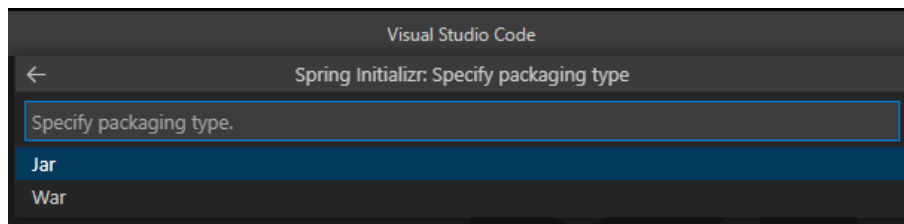


Una vez instaladas las anteriores extensiones, se puede proceder a crear un proyecto.

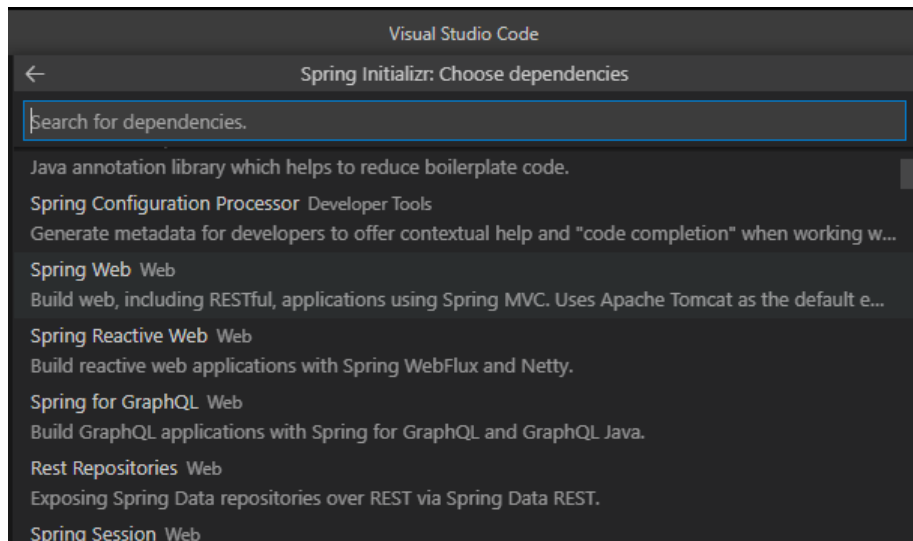
La extensión *Spring Initializr* le permite buscar dependencias y generar nuevos proyectos *Spring Boot*.

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:

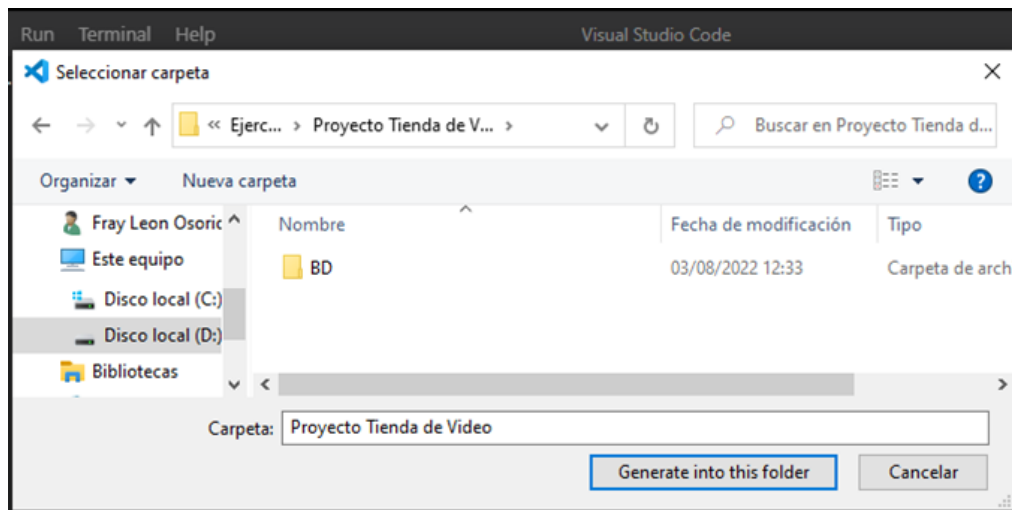




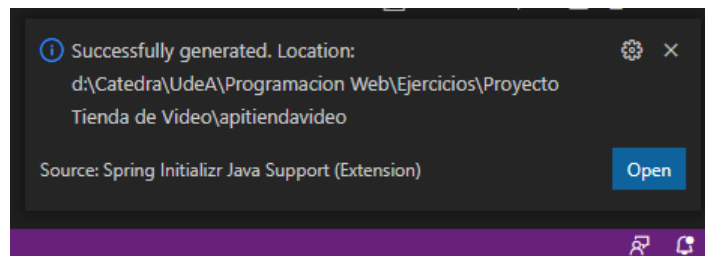
Cuando se eligen las dependencias, en el caso de un proyecto *API REST* se debe elegir **Spring Web**:



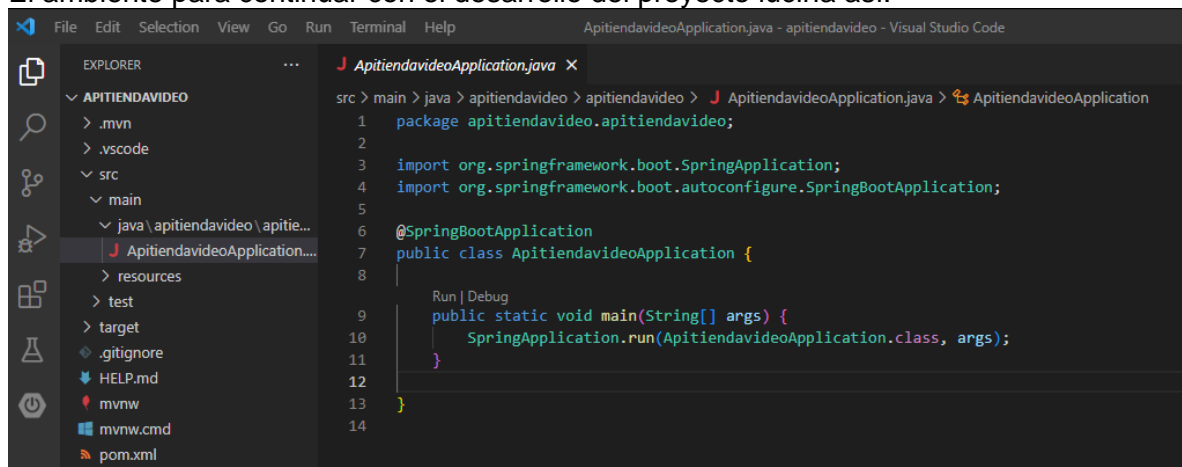
El asistente termina con la definición de la ruta donde se alojará el proyecto:



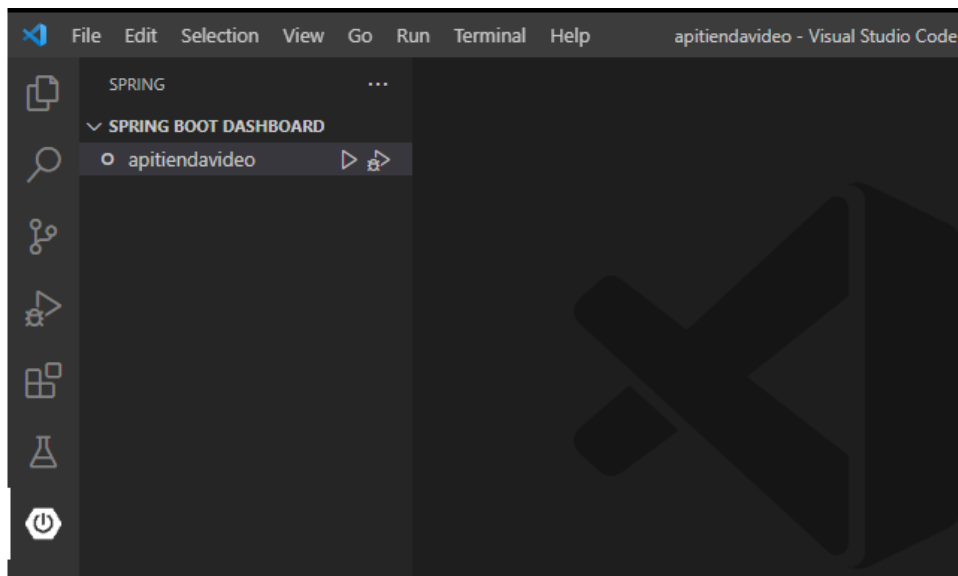
Se procede por tanto a generar el proyecto en la carpeta, y cuando termine, sale una ventana de mensaje con el proceso de generación exitoso y un botón para abrirlo en VS Code:



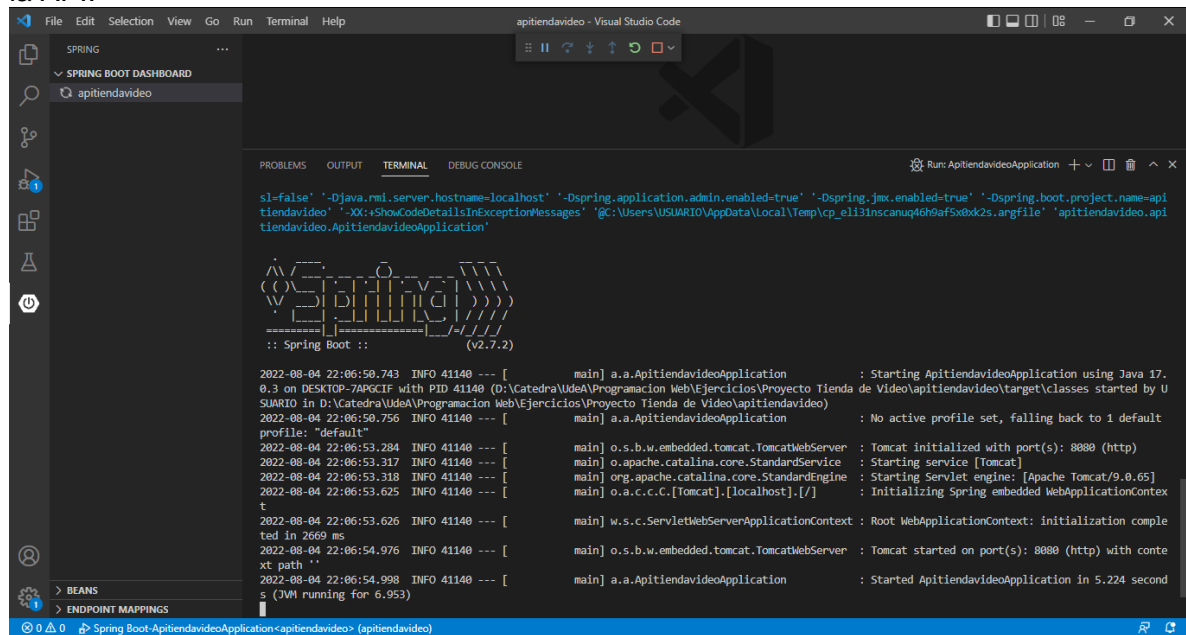
El ambiente para continuar con el desarrollo del proyecto luciría así:



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*.

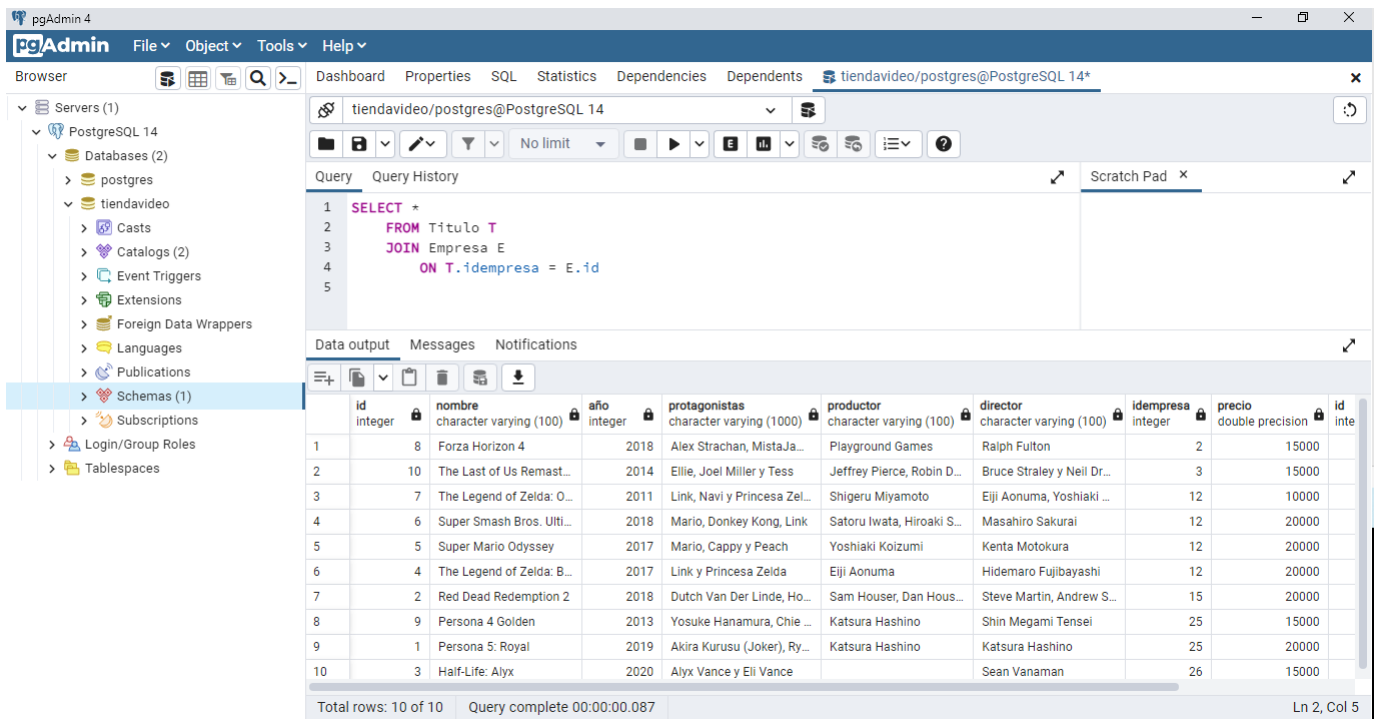
Además de ser un sistema de código abierto es gratuito y multiplataforma, y su desarrollo es llevado adelante por una gran comunidad de colaboradores de todo el mundo que día a día ponen su granito de arena para hacer de este sistema una de las opciones más sólidas a nivel de bases de datos.

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. On the left, the 'Servers' tree is expanded to show the 'tiendavideo' database. The main pane displays a SQL query in the 'Query' tab:

```

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

```

Below the query, the 'Data output' tab shows the results of the query. The table has 10 rows and 10 columns. The columns are: id (integer), nombre (character varying (100)), año (integer), protagonistas (character varying (1000)), productor (character varying (100)), director (character varying (100)), idempresa (integer), precio (double precision), and id (integer). The results are as follows:

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 Kurosawa (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**).

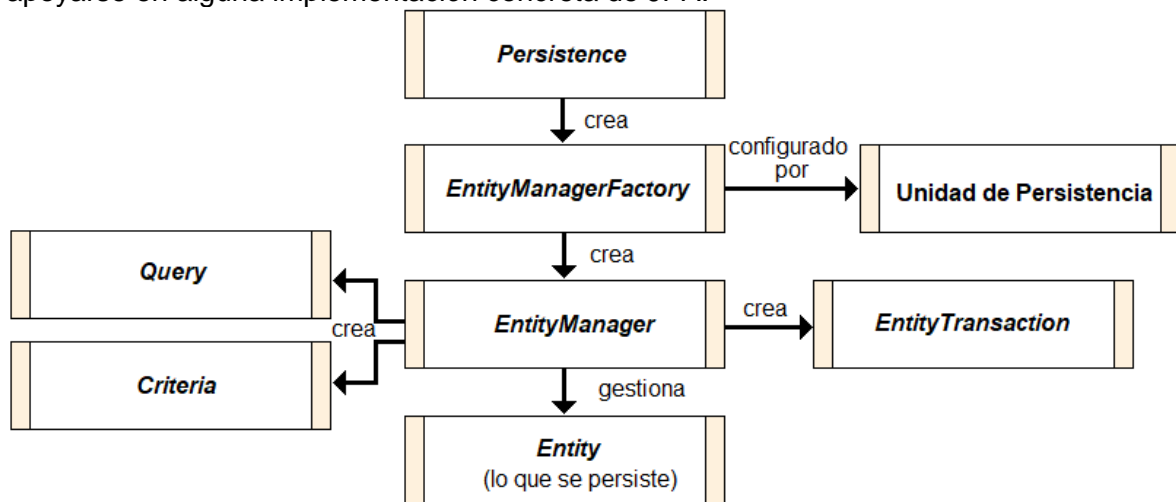
Con lo anterior en mente, se puede definir entonces que es la **API de Persistencia de Java** o **JPA**.

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 javax.persistence.*;

@Entity
@Table(name = "empresa")
public class Empresa {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private long id;

    @ManyToOne
    @JoinColumn(name = "idpais", referencedColumnName = "id")
    private Pais pais;

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

    public Empresa() {
    }

    public long getId() {
        return id;
    }

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

    public Empresa(long id, Pais pais, String nombre) {
        this.id = id;
        this.pais = pais;
        this.nombre = nombre;
    }
}
```

```
}

public Pais getPais() {
    return pais;
}

public void setPais(Pais pais) {
    this.pais = pais;
}

public String getNombre() {
    return nombre;
}

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

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		
@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.

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.url=jdbc:postgresql://localhost:5432/campeonatosfifa
spring.datasource.username=postgres
spring.datasource.password=password
```

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 pais (pais, entidad)
VALUES ('Polonia', 'Asociación Polaca de Futbol')";
        int rows = jdbcTemplate.update(sql);
        if (rows > 0) {
            System.out.println("Se agregó una nueva selección.");
        }
    }
}
```

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.PostgreSQLDialect
```

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 apicampeonatosfifa.apicampeonatosfifa.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 = "pais")
public class Seleccion {

    @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 = "entidad", length = 100, unique = true)
    private String entidad;

    public Seleccion() {
    }

    public Seleccion(long id, String nombre, String entidad) {
        this.id = id;
        this.nombre = nombre;
        this.entidad = entidad;
    }

    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 getEntidad() {  
    return entidad;  
}  
  
public void setEntidad(String entidad) {  
    this.entidad = entidad;  
}  
}
```

En este código se mapeó la tabla *Pais* como la entidad *Seleccion*.

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

```
package apicampeonatosfifa. apicampeonatosfifa.repositorio;  
  
import org.springframework.data.jpa.repository.JpaRepository;  
import org.springframework.stereotype.Repository;  
  
import apicampeonatosfifa. apicampeonatosfifa.modelo.Pais;  
  
@Repository  
public interface Seleccion Repositorio extends JpaRepository<  
    Seleccion, Long>{  
  
}
```

Y luego se puede usar este repositorio en un controlador *Spring MVC* o clase empresarial de la siguiente manera:

```
package apicampeonatosfifa. apicampeonatosfifa.controlador;  
  
import java.util.*;  
  
import org.springframework.beans.factory.annotation.*;  
import org.springframework.web.bind.annotation.*;  
  
import apicampeonatosfifa. apicampeonatosfifa.modelo.*;  
import apicampeonatosfifa. apicampeonatosfifa.repositorio.*;  
  
@RestController  
@RequestMapping("/selecciones")  
public class SeleccionControlador {
```

```
@Autowired
private SeleccionRepositorio repositorio;

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

En este código se plantea una consulta a la base de datos de todos los registros de la tabla *Pais* mediante un método *REST* definido por la ruta:

[/selecciones/listar](#)

¿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).

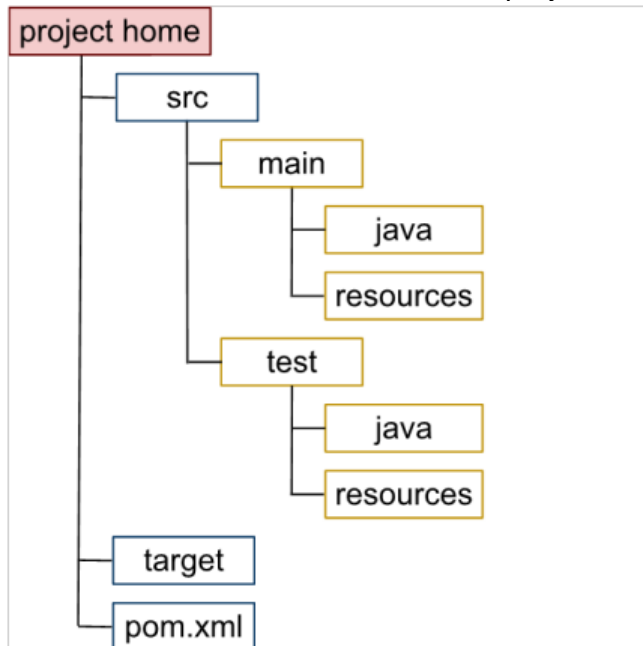
Se trata de un archivo XML llamado *pom.xml* que se encuentra por defecto en la raíz de los proyectos y que contiene toda la información del proyecto: su configuración, sus dependencias, etc...

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 *Campeonato*, se podría realizar la siguiente **Solicitud (Request)** en inglés:

Verbo

Url

GET Listar Campeonatos

+

...

No Environment

HTTP

API Campeonatos FIFA / Listar Campeonatos

Save

GET

http://localhost:8080/campeonatos/listar

Send

Params

Authorization

Headers (6)

Body

Pre-request Script

Tests

Settings

Cookies

Query Params

Key	Value	Description	...	Bulk Edit
Key	Value	Description		

Body

Cookies

Headers (8)

Test Results

Status: 200 OK

Time: 65 ms

Size: 1.54 KB

Save as Example

...

Pretty

Raw

Preview

Visualize

JSON

```

1  [
2    {
3      "id": 1,
4      "nombre": "FIFA World Cup 2010",
5      "pais": {
6        "id": 3,
7        "nombre": "Sudáfrica",
8        "entidad": "Sin Entidad"
9      },
10     "año": 2010
11   },
12   {
13     "id": 2,
14     "nombre": "FIFA World Cup 2006",
15     "pais": {
16       "id": 4,
17       "nombre": "Alemania",

```

Solicitud (Request)

Respuesta (Response)

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*):

```

PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE
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 *“/campeonatos”* y *“/listar”* (mediante la anotación **@RequestMapping** en el controlador).

En consecuencia, la Url de la solicitud queda así:

```
http://localhost:8080/campeonatos/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": "FIFA World Cup 2010",
    "pais": {
      "id": 3,
      "nombre": "Sudáfrica",
      "entidad": "Sin Entidad"
    },
    "año": 2010
  },
  {
    "id": 2,
    "nombre": "FIFA World Cup 2006",
    "pais": {
      "id": 4,
      "nombre": "Alemania",
      "entidad": "Federación Alemana de Futbol"
    },
    "año": 2006
  },
  {
    "id": 3,
    "nombre": "FIFA World Cup 2002",
    "pais": {
      "id": 5,
      "nombre": "Japón",
      "entidad": "Sin Entidad"
    },
    "año": 2002
  },
  {
    "id": 4,
    "nombre": "FIFA World Cup 1998",
    "pais": {
      "id": 6,
      "nombre": "Francia",

```



```
        "entidad": "Federación de Fútbol de Francia"
    },
    "año": 1998
},
{
    "id": 5,
    "nombre": "FIFA World Cup 1994",
    "pais": {
        "id": 7,
        "nombre": "Estados Unidos",
        "entidad": "Sin Entidad"
    },
    "año": 1994
},
{
    "id": 6,
    "nombre": "FIFA World Cup 1990",
    "pais": {
        "id": 8,
        "nombre": "Italia",
        "entidad": "Sin Entidad"
    },
    "año": 1990
},
{
    "id": 7,
    "nombre": "FIFA World Cup U-20 2011",
    "pais": {
        "id": 1,
        "nombre": "Colombia",
        "entidad": "Federación Colombiana de Fútbol"
    },
    "año": 2011
},
{
    "id": 8,
    "nombre": "FIFA World Cup U-20 2009",
    "pais": {
        "id": 45,
        "nombre": "Egipto",
        "entidad": "Sin Entidad"
    },
    "año": 2009
},
{
    "id": 9,
    "nombre": "FIFA World Cup 2014",
    "pais": {
        "id": 11,
        "nombre": "Brasil",
        "entidad": "Confederación Brasileña de Fútbol"
    },
    "año": 2014
}
```




```
    },  
    {  
      "id": 10,  
      "nombre": "FIFA World Cup 2018",  
      "pais": {  
        "id": 49,  
        "nombre": "Rusia",  
        "entidad": "Sin Entidad"  
      },  
      "año": 2018  
    },  
    {  
      "id": 11,  
      "nombre": "FIFA World Cup 2022",  
      "pais": {  
        "id": 60,  
        "nombre": "Qatar",  
        "entidad": "Sin Entidad"  
      },  
      "año": 2012  
    }  
  ]
```

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 *apicampeonatosfifa* y que contenga las siguientes dependencias:

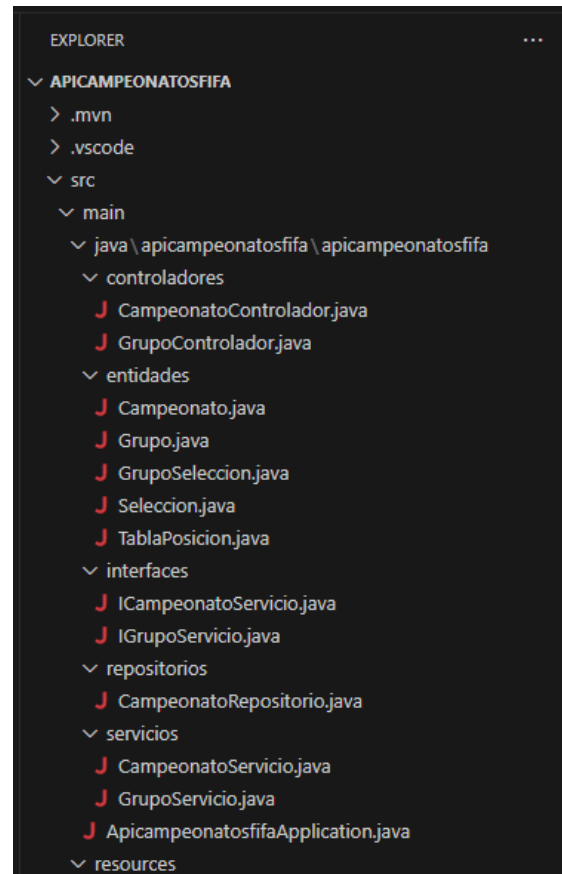
Grupo	Artefacto
org.springframework.boot	spring-boot-starter-web
	spring-boot-starter-data-jpa
	spring-boot-devtools
org.postgresql	postgresql

Acorde con las arquitecturas limpias, se plantea la siguiente distribución de las clases que componen el proyecto:

- Paquete *apicampeonatosfifa.apicampeonatosfifa.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 *apicampeonatosfifa.apicampeonatosfifa.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*.

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 `apicampeonatosfifa.apicampeonatosfifa.interfaces` con las clases correspondientes a las interfaces propias del proyecto. 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.

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 `apicampeonatosfifa.apicampeonatosfifa.servicios` 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 la 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 CampeonatoServicio implements ICampeonatoServicio {

    private CampeonatoRepositorio repositorio;

    public CampeonatoServicio(CampeonatoRepositorio repositorio) {
        this.repositorio = repositorio;
    }

    ...
}
```

Aquí en el método constructor de la clase se inyecta el objeto del 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*).

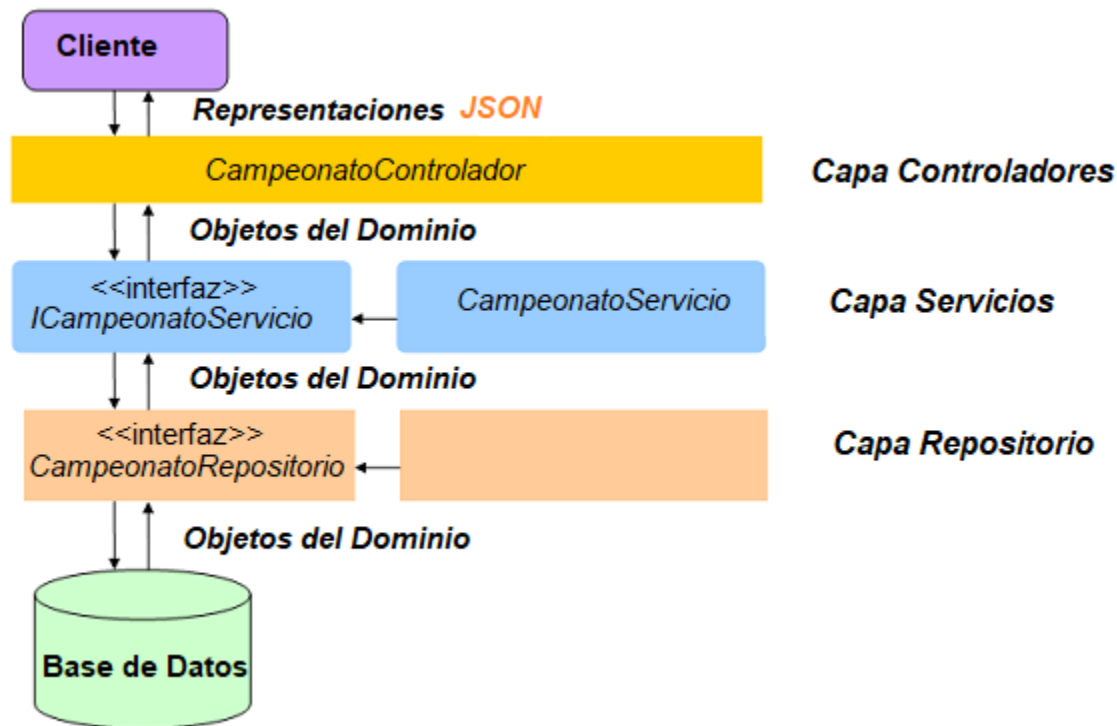
Spring Boot también permite la inyección de las dependencias a través de anotaciones como **@Autowired**

- Paquete *apicampeonatosfifa.apicampeonatosfifa.controladores* 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 *Campeonatos*:



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 *Campeonato*

```
package apicampeonatosfifa.apicampeonatosfifa.entidades;

import java.util.ArrayList;
import java.util.List;

import org.hibernate.annotations.GenericGenerator;

import com.fasterxml.jackson.annotation.JsonIgnore;

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.OneToMany;
import jakarta.persistence.Table;

@Entity
@Table(name = "campeonato")
public class Campeonato {
    @Id
    @Column(name = "id")
    @GeneratedValue(strategy = GenerationType.AUTO, generator =
"secuencia_campeonato")
    @GenericGenerator(name = "secuencia_campeonato", strategy =
"increment")
    private long id;

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

    @ManyToOne
    @JoinColumn(name = "idpais", referencedColumnName = "id")
    private Seleccion pais;

    @Column(name = "año")
    private int año;

    @JsonIgnore
    @OneToMany(mappedBy = "campeonato")
    private List<Grupo> grupos = new ArrayList<>();

    public Campeonato() {
    }

    public Campeonato(long id, String nombre, Seleccion pais, int
año) {
        this.id = id;
        this.nombre = nombre;
    }
}
```



```
        this.pais = pais;
        this.año = año;
    }

    public Campeonato(long id, String nombre, Seleccion pais, int
año, List<Grupo> grupos) {
        this.id = id;
        this.nombre = nombre;
        this.pais = pais;
        this.año = año;
        this.grupos = grupos;
    }

    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 int getAño() {
        return año;
    }

    public void setAño(int año) {
        this.año = año;
    }

    public Seleccion getPais() {
        return pais;
    }

    public void setPais(Seleccion pais) {
        this.pais = pais;
    }

    public List<Grupo> getGrupos() {
        return grupos;
    }

    public void setGrupos(List<Grupo> grupos) {
        this.grupos = grupos;
    }
}
```

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 *Seleccion* (es el alias que tendrá la tabla *Pais* de la base de datos)

```
package apicampeonatosfifa.apicampeonatosfifa.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 = "pais")
public class Seleccion {

    @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 = "entidad", length = 100, unique = true)
    private String entidad;

    public Seleccion() {
    }

    public Seleccion(long id, String nombre, String entidad) {
        this.id = id;
        this.nombre = nombre;
        this.entidad = entidad;
    }

    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 getEntidad() {
    return entidad;
}

public void setEntidad(String entidad) {
    this.entidad = entidad;
}
}
```

- **Clase Grupo**

```
package apicampeonatosfifa.apicampeonatosfifa.entidades;

import java.util.ArrayList;
import java.util.List;

import org.hibernate.annotations.GenericGenerator;

import com.fasterxml.jackson.annotation.JsonIgnore;

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.OneToMany;
import jakarta.persistence.Table;

@Entity
@Table(name = "grupo")
public class Grupo {

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



```
@ManyToOne
@JoinColumn(name = "idcampeonato", referencedColumnName = "id")
private Campeonato campeonato;

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

@JsonIgnore
@OneToMany(mappedBy = "grupo")
private List<GrupoSeleccion> selecciones = new ArrayList<>();

public Grupo() {
}

public Grupo(long id, Campeonato campeonato, String nombre,
List<GrupoSeleccion> selecciones) {
    this.id = id;
    this.campeonato = campeonato;
    this.nombre = nombre;
    this.selecciones = selecciones;
}

public Grupo(long id, Campeonato campeonato, String nombre) {
    this.id = id;
    this.campeonato = campeonato;
    this.nombre = nombre;
}

public long getId() {
    return id;
}

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

public Campeonato getCampeonato() {
    return campeonato;
}

public void setCampeonato(Campeonato campeonato) {
    this.campeonato = campeonato;
}

public String getNombre() {
    return nombre;
}

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

```
public List<GrupoSeleccion> getSelecciones() {  
    return selecciones;  
}  
  
public void setSelecciones(List<GrupoSeleccion> selecciones) {  
    this.selecciones = selecciones;  
}  
}
```

De manera similar a la clase *Campeonato*, tiene una lista de entidades *GrupoSeleccion* para representar la relación de las selecciones que tiene el grupo.

- Clase *GrupoSeleccion* (es el alias que tendrá la tabla *GrupoPais* de la base de datos

```
package apicampeonatosfifa.apicampeonatosfifa.entidades;  
  
import jakarta.persistence.Entity;  
import jakarta.persistence.Id;  
import jakarta.persistence.JoinColumn;  
import jakarta.persistence.ManyToOne;  
import jakarta.persistence.Table;  
  
@Entity  
@Table(name = "grupopais")  
public class GrupoSeleccion {  
  
    @Id  
    @ManyToOne  
    @JoinColumn(name = "idgrupo", referencedColumnName = "id")  
    private Grupo grupo;  
  
    @Id  
    @ManyToOne  
    @JoinColumn(name = "idpais", referencedColumnName = "id")  
    private Seleccion seleccion;  
  
    public GrupoSeleccion() {  
    }  
  
    public GrupoSeleccion(Gruo grupo, Seleccion seleccion) {  
        this.grupo = grupo;  
        this.seleccion = seleccion;  
    }  
  
    public Grupo getGrupo() {  
        return grupo;  
    }  
  
    public void setGrupo(Gruo grupo) {  
        this.grupo = grupo;  
    }  
}
```

```
public Seleccion getSeleccion() {  
    return seleccion;  
}  
  
public void setSeleccion(Seleccion seleccion) {  
    this.seleccion = seleccion;  
}  
}
```

En esta clase en particular, la clave primaria es la combinación de 2 campos, los cuales a su vez son claves foráneas

- Clase *TablaPosicion* (con la estructura de la información que se recibirá de la función que obtiene la tabla de posiciones

```
package apicampeonatosfifa.apicampeonatosfifa.entidades;  
  
import jakarta.persistence.Column;  
import jakarta.persistence.Entity;  
import jakarta.persistence.Id;  
  
@Entity  
public class TablaPosicion {  
  
    @Id  
    @Column(name = "id")  
    private Long id;  
    @Column(name = "pais")  
    private String pais;  
    @Column(name = "pJ")  
    private int pJ;  
    @Column(name = "pG")  
    private int pG;  
    @Column(name = "pE")  
    private int pE;  
    @Column(name = "pP")  
    private int pP;  
    @Column(name = "gF")  
    private int gF;  
    @Column(name = "gC")  
    private int gC;  
    @Column(name = "puntos")  
    private int puntos;  
  
    public TablaPosicion() {  
    }  
  
    public TablaPosicion(Long id, String pais, int pJ, int pG, int  
pE, int pP, int gF, int gC, int puntos) {
```



```
        this.id = id;
        this.pais = pais;
        this.pJ = pJ;
        this.pG = pG;
        this.pE = pE;
        this.pP = pP;
        this.gF = gF;
        this.gC = gC;
        this.puntos = puntos;
    }

    public String getPais() {
        return pais;
    }

    public int getPJ() {
        return pJ;
    }

    public int getPJ() {
        return pG;
    }

    public int getPJ() {
        return pE;
    }

    public int getPJ() {
        return pP;
    }

    public int getGF() {
        return gF;
    }

    public int getGC() {
        return gC;
    }

    public int getPuntos() {
        return puntos;
    }
}
```

Repositorios

Para este ejercicio sólo se requerirá un repositorio.

- Interfaz *CampeonatoRepositorio* que define los métodos para operar con la entidad *Campeonato*

```
package apicampeonatosfifa.apicampeonatosfifa.repositorios;

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

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

import apicampeonatosfifa.apicampeonatosfifa.entidades.Campeonato;

@Repository
public interface CampeonatoRepositorio extends
JpaRepository<Campeonato, Long> {

}
```

Contratos para los Servicios

Se definen los siguientes contratos para los servicios:

- Interfaz *ICampeonatoServicio* que incluye los métodos para listar todos los campeonatos y listar los grupos de un campeonato:

```
package apicampeonatosfifa.apicampeonatosfifa.interfaces;

import java.util.List;

import apicampeonatosfifa.apicampeonatosfifa.entidades.Campeonato;
import apicampeonatosfifa.apicampeonatosfifa.entidades.Grupo;

public interface ICampeonatoServicio {

    public List<Campeonato> listar();

    public List<Grupo> listarGrupos(long id);

}
```

- Interfaz *IGrupoServicio* que incluye el método para obtener la tabla de posiciones de un grupo

```
package apicampeonatosfifa.apicampeonatosfifa.interfaces;

import java.util.List;

import apicampeonatosfifa.apicampeonatosfifa.entidades.TablaPosicion;

public interface IGrupoServicio {

    public List<TablaPosicion> obtenerPosiciones(int id);

}
```

Servicios

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

- Clase *CampeonatoServicio*

```
package apicampeonatosfifa.apicampeonatosfifa.servicios;
```

```
import java.util.List;
import org.springframework.stereotype.Service;
import apicampeonatosfifa.apicampeonatosfifa.entidades.Campeonato;
import apicampeonatosfifa.apicampeonatosfifa.entidades.Grupo;
import
apicampeonatosfifa.apicampeonatosfifa.interfaces.ICampeonatoServicio;
import
apicampeonatosfifa.apicampeonatosfifa.repositorios.CampeonatoRepositorio;

@Service
public class CampeonatoServicio implements ICampeonatoServicio {

    private CampeonatoRepositorio repositorio;

    public CampeonatoServicio(CampeonatoRepositorio repositorio) {
        this.repositorio = repositorio;
    }

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

    @Override
    public List<Grupo> listarGrupos(long id) {
        // Buscar campeonato por su ID
        var campeonatoBuscado = repositorio.findById(id);

        if (campeonatoBuscado.isPresent()) {
            var campeonato = campeonatoBuscado.get();

            return campeonato.getGrupos();
        }
        return null;
    }
}
```

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 *GrupoServicio*:**

```
package apicampeonatosfifa.apicampeonatosfifa.servicios;

import java.util.List;
import org.springframework.stereotype.Service;
import apicampeonatosfifa.apicampeonatosfifa.entidades.TablaPosicion;
import apicampeonatosfifa.apicampeonatosfifa.interfaces.IGrupoServicio;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;

@Service
public class GrupoServicio implements IGrupoServicio {

    @PersistenceContext
    public EntityManager em;

    @Override
    public List<TablaPosicion> obtenerPosiciones(int idGrupo) {
        List<TablaPosicion> tablaPosiciones = em
            .createNativeQuery("SELECT * FROM fTablaPosicionesGrupo(:idgrupotabla)", TablaPosicion.class)
            .setParameter("idgrupotabla", idGrupo)
            .getResultList();
        return tablaPosiciones;
    }
}
```

En ese caso para ejecutar la consulta a la base de datos en lugar de hacerlo mediante un repositorio, se procede a utilizar la interfaz ***EntityManager*** la cual forma parte de la API de persistencia de Java. Esta se utiliza para administrar entidades (objetos Java) y realizar operaciones de persistencia en una base de datos relacional permitiendo crear, leer, actualizar y eliminar dichas entidades. También permite operar con objetos de la base de datos como procedimientos almacenados y funciones, así como ejecutar consultas en el lenguaje nativo del motor de base de datos.

Se debe observar que, para poder realizar la inyección de dependencias de implementaciones de esta interfaz, se requiere la anotación ***@PersistenceContext***.

En este ejercicio, se va a llamar la función de tabla *fTablaPosicionesGrupo* de la base de datos mediante una instrucción en el lenguaje de consulta de *PostgreSQL*

pasada como parámetro del método ***createNativeQuery()*** de la instancia de *EntityManager*. Este método define la consulta y para pasar los parámetros de la función de tabla se requiere adicionalmente del método ***setParameter()*** y así mismo para obtener los resultados, se invoca después el método ***getResultList()***. Estos métodos se deben llamar de forma encadenada

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*.

El uso de la anotación *@CrossOrigin* permite que el método sea accedido desde aplicaciones clientes web desarrollados en frameworks como *Angular CLI*.

- Clase *CampeonatoControlador* con los métodos web para listar todos los campeonatos y listar los grupos de un campeonato

```
package apicampeonatosfifa.apicampeonatosfifa.controladores;

import java.util.List;

import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import apicampeonatosfifa.apicampeonatosfifa.entidades.Campeonato;
import apicampeonatosfifa.apicampeonatosfifa.entidades.Grupo;
import apicampeonatosfifa.apicampeonatosfifa.interfaces.ICampeonatoServicio;

@RestController
@RequestMapping("/campeonatos")
public class CampeonatoControlador {

    private ICampeonatoServicio servicio;

    public CampeonatoControlador(ICampeonatoServicio servicio) {
        this.servicio = servicio;
    }

    @CrossOrigin(origins = "http://localhost:4200")
    @RequestMapping(value = "/listar", method = RequestMethod.GET)
```

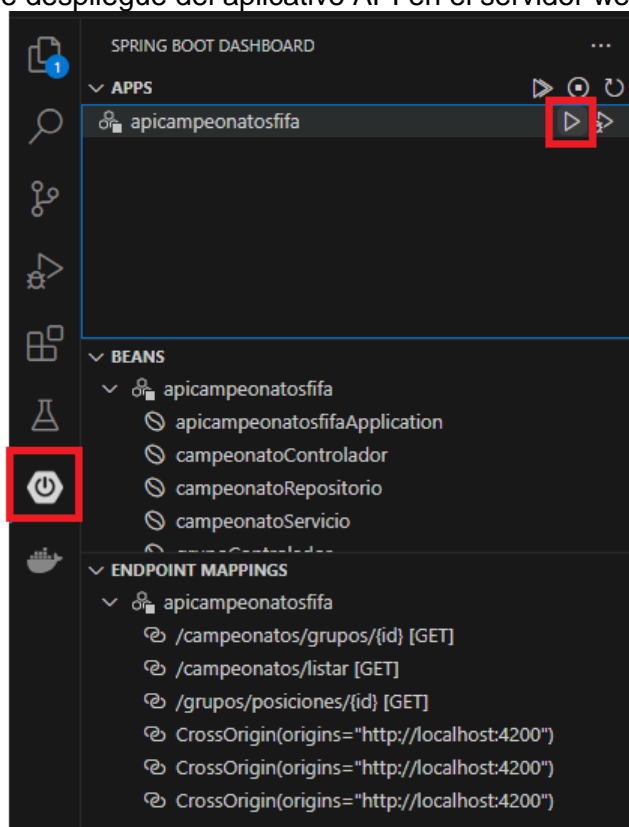
```
public List<Campeonato> listar() {  
    return servicio.listar();  
}  
  
@CrossOrigin(origins = "http://localhost:4200")  
@RequestMapping(value = "/grupos/{id}", method =  
RequestMethod.GET)  
public List<Grupo> listarGrupos(@PathVariable long id) {  
    return servicio.listarGrupos(id);  
}  
}
```

En el caso del método *listarGrupos()* 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*

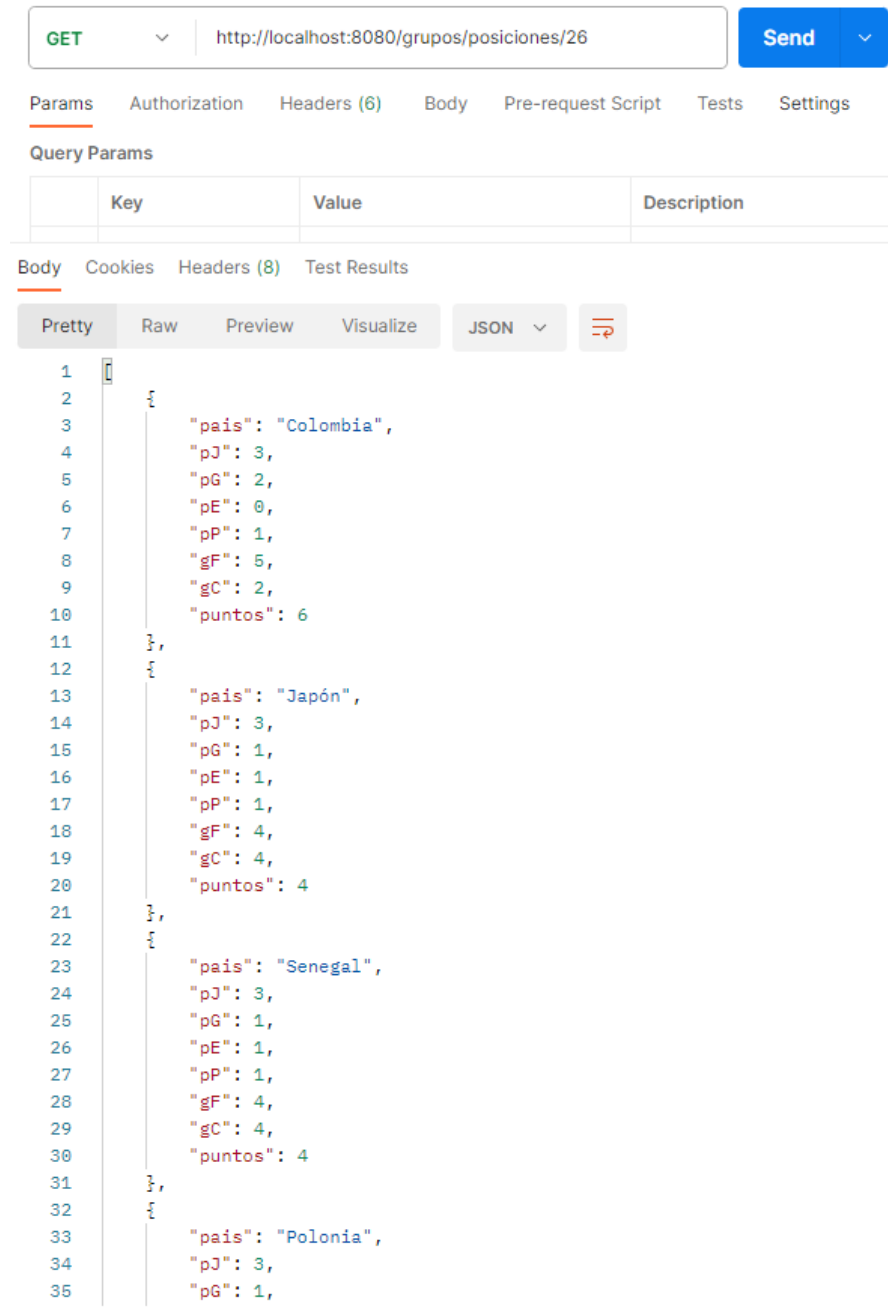
- Clase *GrupoControlador* con el método web para obtener la tabla de posiciones

```
package apicampeonatosfifa.apicampeonatosfifa.controladores;  
  
import java.util.List;  
import org.springframework.web.bind.annotation.CrossOrigin;  
import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RequestMethod;  
import org.springframework.web.bind.annotation.RestController;  
import apicampeonatosfifa.apicampeonatosfifa.entidades.TablaPosicion;  
import apicampeonatosfifa.apicampeonatosfifa.interfaces.IGrupoServicio;  
  
@RestController  
@RequestMapping("/grupos")  
public class GrupoControlador {  
  
    private IGrupoServicio servicio;  
  
    public GrupoControlador(IGrupoServicio servicio) {  
        this.servicio = servicio;  
    }  
  
    @CrossOrigin(origins = "http://localhost:4200")  
    @RequestMapping(value = "/posiciones/{id}", method =  
RequestMethod.GET)  
    public List<TablaPosicion> obtenerPosiciones(@PathVariable int  
id) {  
        return servicio.obtenerPosiciones(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 mediante *Postman*:




GET Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Query Params

Key	Value	Description
-----	-------	-------------

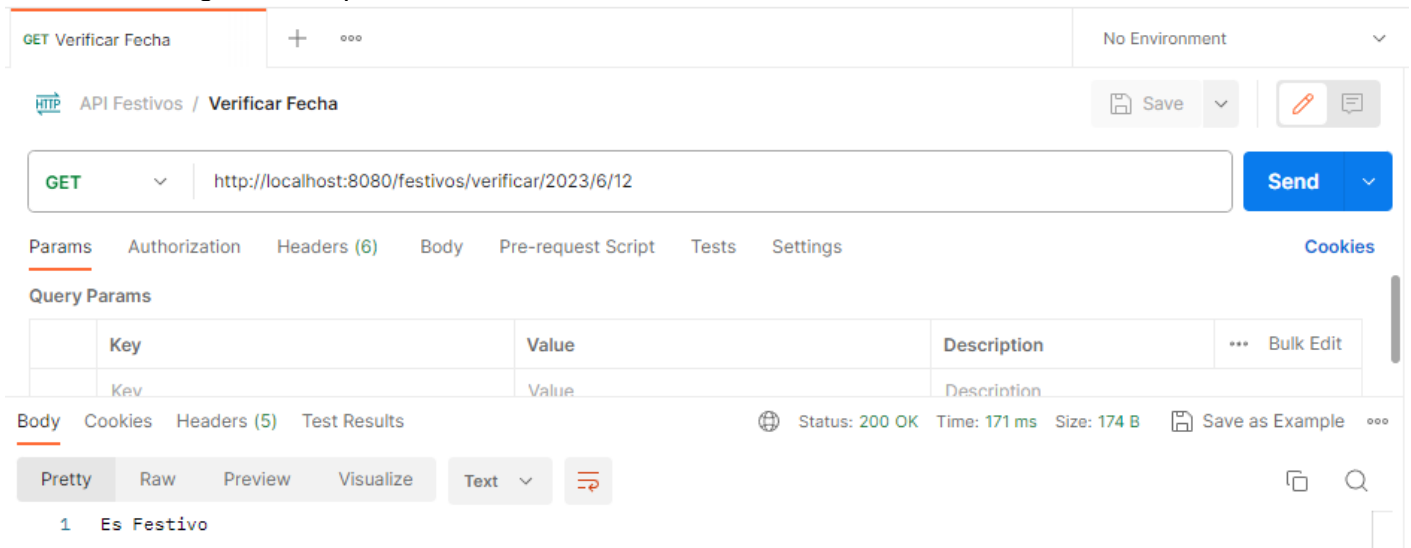
Body Cookies Headers (8) Test Results

Pretty Raw Preview Visualize JSON 

```
1  {
2    "pais": "Colombia",
3    "pJ": 3,
4    "pG": 2,
5    "pE": 0,
6    "pP": 1,
7    "gF": 5,
8    "gC": 2,
9    "puntos": 6
10  },
11  {
12    "pais": "Japón",
13    "pJ": 3,
14    "pG": 1,
15    "pE": 1,
16    "pP": 1,
17    "gF": 4,
18    "gC": 4,
19    "puntos": 4
20  },
21  {
22    "pais": "Senegal",
23    "pJ": 3,
24    "pG": 1,
25    "pE": 1,
26    "pP": 1,
27    "gF": 4,
28    "gC": 4,
29    "puntos": 4
30  },
31  {
32    "pais": "Polonia",
33    "pJ": 3,
34    "pG": 1,
```

2. Elaborar una API REST en *Spring Boot* que incluya un servicio web que permita validar si una fecha determinada es festiva o no.

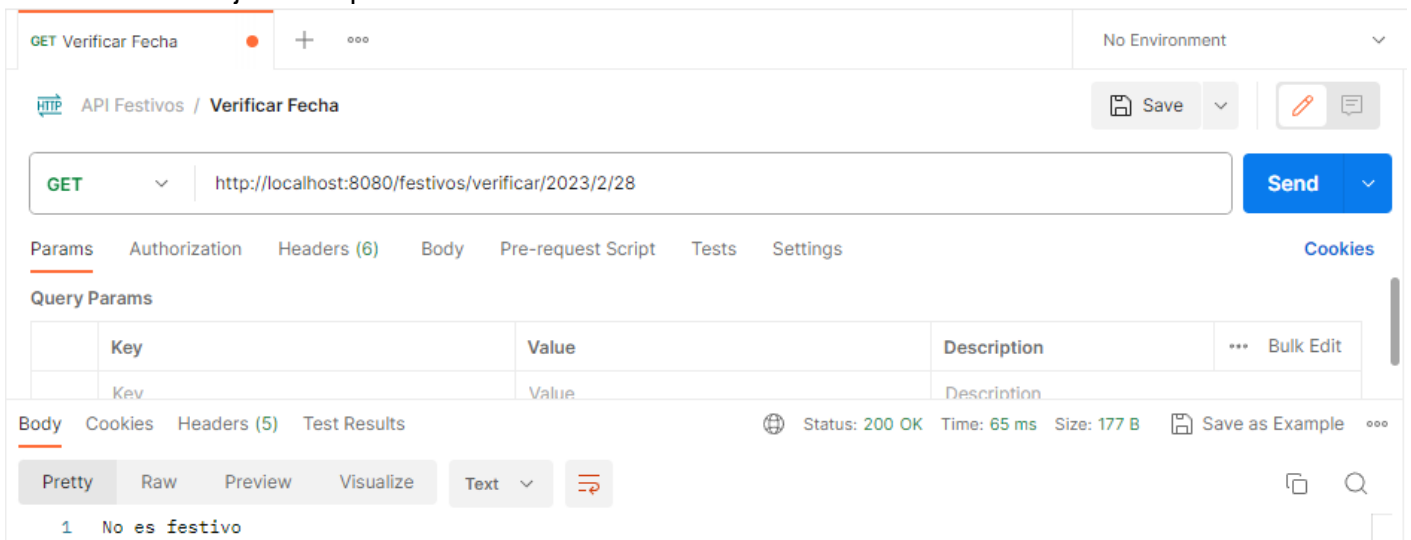
Por ejemplo, la ejecución del servicio para el 12 de junio de 2023, entregaría la siguiente respuesta:



The screenshot shows a Postman interface for a REST client. The request is a GET to `http://localhost:8080/festivos/verificar/2023/6/12`. The response status is 200 OK, and the body is `Es Festivo`.

Key	Value	Description
Key	Value	Description

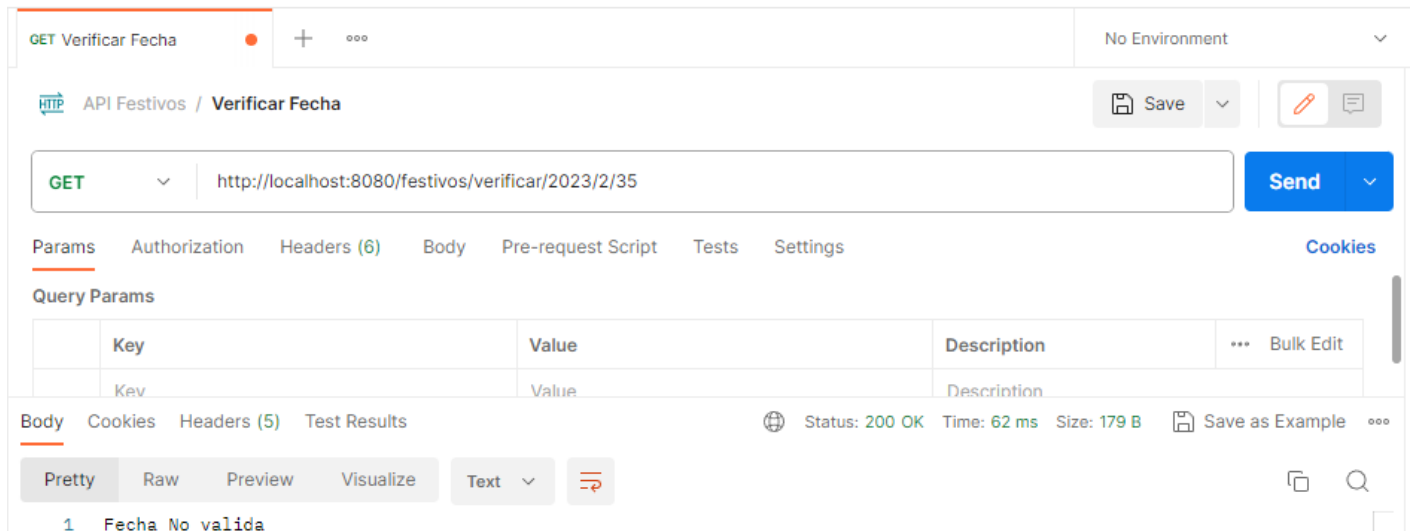
La ejecución para el 28 de febrero de 2023:



The screenshot shows a Postman interface for a REST client. The request is a GET to `http://localhost:8080/festivos/verificar/2023/2/28`. The response status is 200 OK, and the body is `No es festivo`.

Key	Value	Description
Key	Value	Description

Y la ejecución para una fecha no válida como el 35 de febrero de 2023:



The screenshot shows a REST client interface with the following details:

- Request Method:** GET
- URL:** http://localhost:8080/festivos/verificar/2023/2/35
- Status:** 200 OK
- Time:** 62 ms
- Size:** 179 B
- Response Body:** 1 Fecha No valida

Para ello se dispondrá de una tabla con la lista de festivos:

Día	Mes	Nombre	Tipo	Días de Pascua
01	01	Año nuevo	1	
06	01	Santos Reyes	2	
19	03	San José	2	
		Jueves Santo	3	-3
		Viernes Santo	3	-2
		Domingo de Pascua	3	0
01	05	Día del Trabajo	1	
		Ascensión del Señor	4	40
		Corpus Christi	4	61
		Sagrado Corazón de Jesús	4	68
29	06	San Pedro y San Pablo	2	
20	07	Independencia Colombia	1	
07	08	Batalla de Boyacá	1	
15	08	Asunción de la Virgen	2	
12	10	Día de la Raza	2	
01	11	Todos los santos	2	
11	11	Independencia de Cartagena	2	
08	12	Inmaculada Concepción	1	
25	12	Navidad	1	

La lista de tipos de festivos es la siguiente:

Índice	Tipo	Modo de calcularlo
1	Fijo	No se puede variar
2	Ley de "Puente festivo"	Se traslada la fecha al siguiente lunes
3	Basado en el domingo de pascua	La fecha se calcula obteniendo la fecha del domingo de pascua y sumándole los días que correspondan.
4	Basado en el domingo de pascua y Ley de "Puente festivo"	La fecha se calcula obteniendo la fecha del domingo de pascua y sumándole los días que correspondan. La fecha calculada debe ser trasladada al siguiente lunes

Calculo del domingo de pascua

Para conocer el día del año que comienza la Semana Santa (es decir, cuando sería el *Domingo de Ramos*) se tiene la siguiente fórmula que calcula el número de días que deben pasar después del 15 de marzo:

$$\text{Días} = d + (2b + 4c + 6d + 5) \text{ MOD } 7$$

Dónde:

a = Año MOD 19

b = Año MOD 4

c = Año MOD 7

d = (19a + 24) MOD 30

Por ejemplo para el año 1999:

a = 1999 MOD 19 = 4

b = 1999 MOD 4 = 3

c = 1999 MOD 7 = 4

d = (19*4 + 24) MOD 30 = 10

Días = 10 + (2*3 + 4*4 + 6*10 + 5) MOD 7 = 13

Significa que el *Domingo de Ramos* sería el 15 + 13 = 28 de Marzo.

Luego el *Domingo de Pascua* sería 7 días después, o sea, Abril 4.

Marzo 1999						
L	M	M	J	V	S	D
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

Abril 1999						
L	M	M	J	V	S	D
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30		