Testing en Java.

El testing en Java se refiere al proceso de verificar el funcionamiento correcto de una aplicación Java mediante la ejecución de pruebas automatizadas. Estas pruebas están diseñadas para probar diferentes aspectos del código, como la funcionalidad, el rendimiento y la integridad.

Existen varias herramientas y frameworks en Java para realizar pruebas, como JUnit, TestNG, Mockito, etc. Estas herramientas proporcionan un conjunto de API y utilidades que facilitan la creación y ejecución de pruebas automatizadas.

Hay varios tipos de pruebas que se pueden realizar en Java:

¿Qué es Unit Testing?

**Unit testing es una técnica de prueba en la que se evalúan unidades individuales de código para asegurarse de que funcionen correctamente**.

El propósito de las pruebas unitarias es probar la corrección del código aislado.

Los test unitarios son realizados durante el desarrollo del código.

Una unidad puede ser una función, método , procedimiento , módulo u objeto.

Integration Test.

Nos permiten verificar los diferentes componentes del programa o aplicaciòn están unidos e interactúan de manera eficiente.

Por ejemplo . Se presenta la siguiente imagen donde tenemos

¿Què es Junit?

JUnit se trata de un **Framework Open Source para la automatización de las pruebas (tanto unitarias, como de integración) en los proyectos Software**. El framework provee al usuario de herramientas, clases y métodos que le facilitan la tarea de realizar pruebas en su sistema y así asegurar su consistencia y funcionalidad.

Para aplicar test con JUNIT 5 vamos a mostrar el siguiente diagrama de una aplicación de Spring Boot .

Diagrama de una aplicación de Spring Boot.

En una aplicación Spring Boot típica, puedes organizar tu código en capas, como la capa de controlador (Controller), la capa de servicio (Service) y la capa de acceso a datos (Repository) para gestionar entidades como "Usuario". Aquí tienes un ejemplo básico de cómo podrías estructurar estas capas para un CRUD (Crear, Leer, Actualizar, Eliminar) de usuarios.

**Capa de Repositorio (Repository):** Esta capa se encarga de interactuar con la base de datos para realizar operaciones de persistencia.

**Capa de Servicio (Service):** Esta capa contiene la lógica de negocio de tu aplicación y orquesta las operaciones entre el controlador y el repositorio.

**Capa de Controlador (Controller):** Esta capa maneja las solicitudes HTTP, llama a los métodos del servicio y devuelve las respuestas adecuadas.

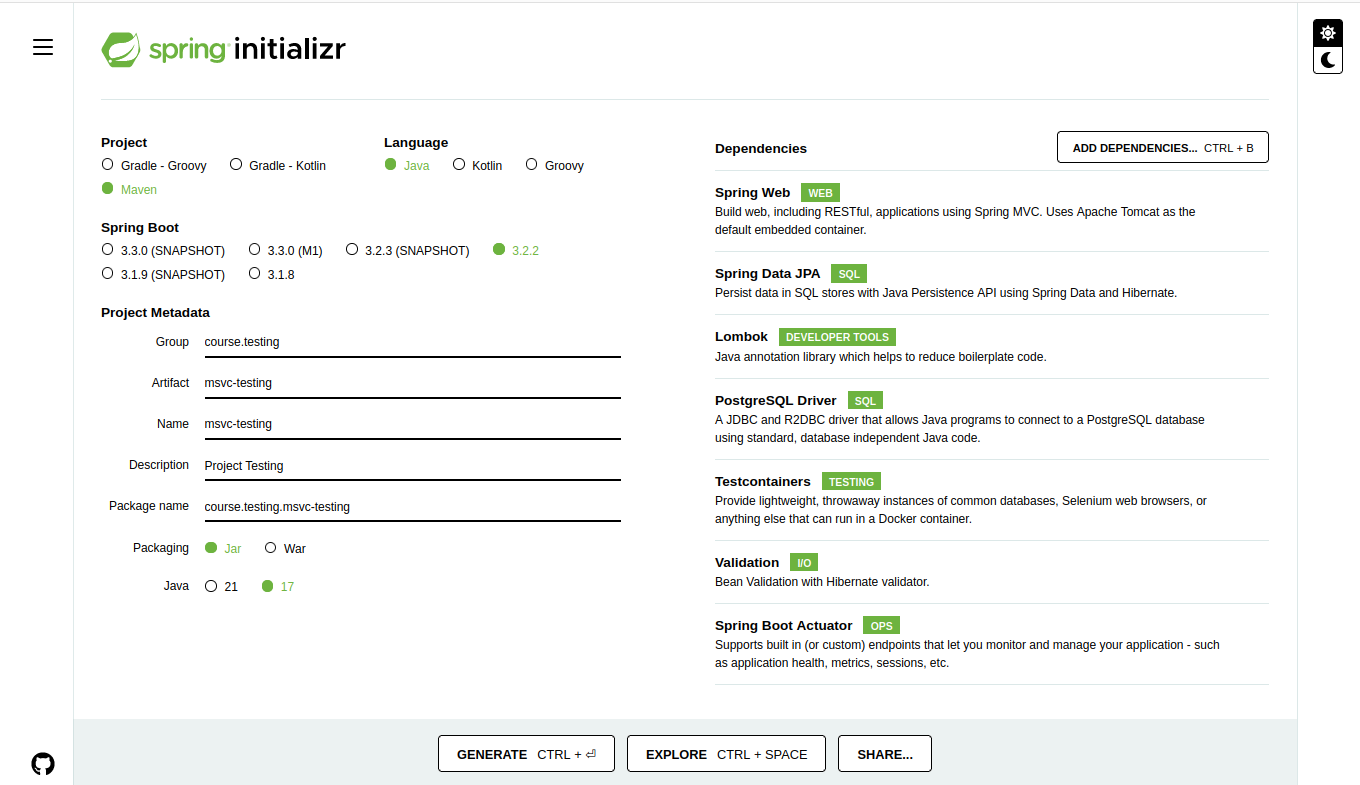
***Testing Spring Boot Application***

* ***Spring***: Es un marco de trabajo para Java que facilita el desarrollo de aplicaciones empresariales, proporcionando herramientas para la gestión de dependencias, configuración, seguridad y más, basado en el principio de inversión de control.
* ***Spring Boot***: Es un proyecto de Spring que simplifica el desarrollo de aplicaciones Spring, eliminando la configuración manual y proporcionando características como autoconfiguración inteligente y un inicio rápido, lo que permite crear aplicaciones Spring con menos esfuerzo y tiempo.

Actualmente se encuentra la versión de Spring es 6 and Spring Boot 3.2.

1 - Creaciòn de Project.

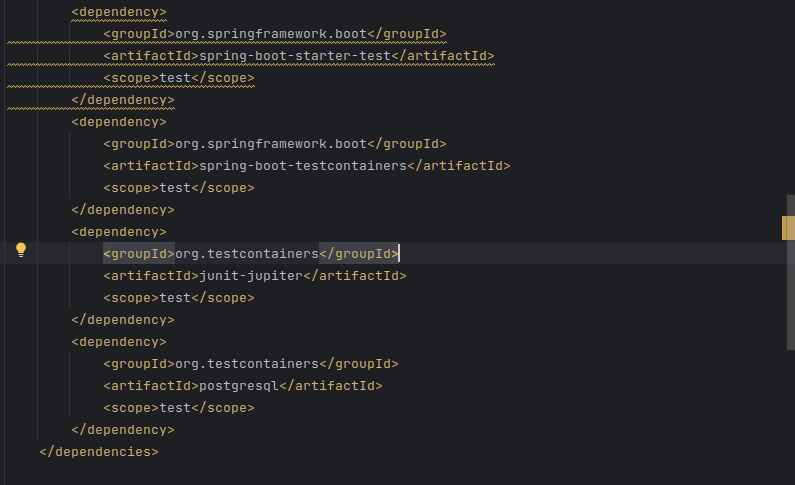
Vamos a la crear el proyecto de testing redireccionandonos a la url siguiente : ***https://start.spring.io/***



Modules a cargar .

* Spring Web:
* Spring Data JPA
* Lombok
* Postgresql Driver
* TestContainers
* Validation
* Spring Boot Actuator.

Revisando el archivo de pom.xml podemos observar :



***Spring Boot Starter Test*** es la dependencia primaria de Testing para Spring Applications. Tiene los siguientes elementos :

* [JUnit](http://junit.org) — The de-facto standard for unit testing Java applications.
* [Spring Test](https://docs.spring.io/spring/docs/4.3.11.RELEASE/spring-framework-reference/htmlsingle/#integration-testing) & Spring Boot Test — Utilities and integration test support for Spring Boot applications.
* [AssertJ](http://joel-costigliola.github.io/assertj/) — A fluent assertion library.
* [Hamcrest](http://hamcrest.org/JavaHamcrest/) — A library of matcher objects (also known as constraints or predicates).
* [Mockito](http://mockito.org/) — A Java mocking framework.
* [JSONassert](https://github.com/skyscreamer/JSONassert) — An assertion library for JSON.
* [JsonPath](https://github.com/jayway/JsonPath) — XPath for JSON.

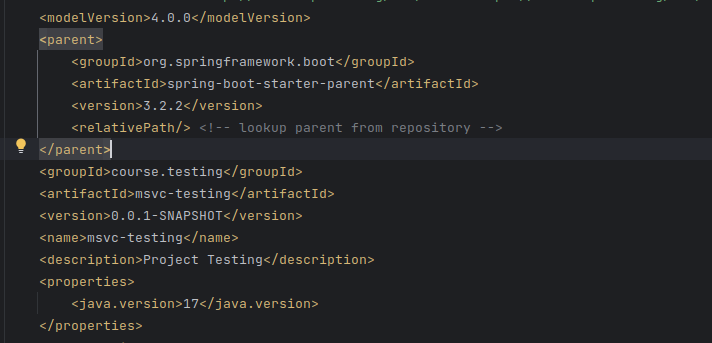
Referencia : <https://docs.spring.io/spring-framework/reference/testing.html#testing>

Descripciòn de cada uno:

* **JUnit**: Marco estándar para pruebas unitarias en Java. Facilita escribir y ejecutar pruebas con anotaciones.
* **Spring Test & Spring Boot Test**: Utilidades para pruebas de integración en aplicaciones Spring y Spring Boot. Permiten probar componentes de Spring como controladores y servicios.
* **AssertJ**: Biblioteca de aserciones fluida que hace que las pruebas unitarias en Java sean más legibles y expresivas.
* **Hamcrest**: Biblioteca para escribir expresiones legibles en pruebas unitarias, proporcionando una amplia gama de matchers predefinidos.
* **Mockito**: Marco de simulación para Java, útil para crear objetos simulados en pruebas unitarias.
* **JSONassert**: Biblioteca para realizar pruebas sobre datos JSON en Java, facilitando la comparación y validación de objetos JSON.
* **JsonPath**: Biblioteca que proporciona una sintaxis similar a XPath para acceder y manipular datos en documentos JSON, útil en pruebas de integración y análisis de datos.

Al seleccionar ***spring-boot-starter-test*** se puede observar las dependencias anteriores.

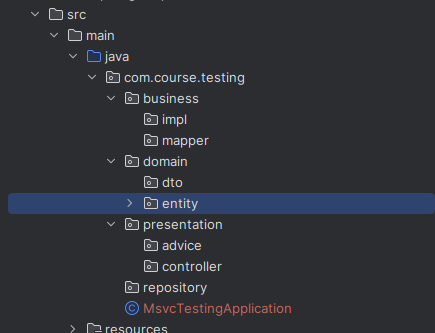
Quien maneja estas dependencias y las versiones es la dependencia padre :



La versión de Spring Boot es ***3.2.2***

1 - **Creación de Estructura del Proyecto.**

A continuaciòn creamos la siguiente estructura de packages :



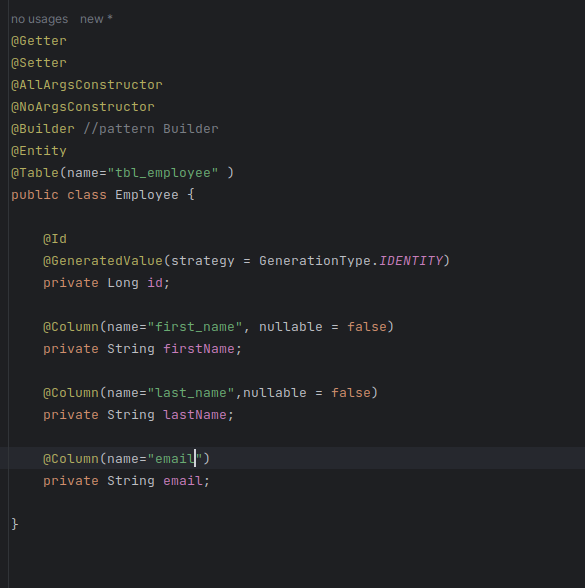
2 - **Create JPA Entity** .

Comenzamos con la creación de la Entity(Entidad**).En Spring Boot, una "entity" se refiere a una clase de Java que representa una entidad de dominio en una aplicación. Estas entidades suelen mapearse a tablas en una base de datos relacional, donde cada instancia de la entidad representa una fila en la tabla.**

Las entidades en Spring Boot se utilizan comúnmente junto con el módulo de persistencia de Spring, como Spring Data JPA e Hibernate, para realizar operaciones de persistencia en la base de datos. Además incorporaremos annotations(anotaciones) de la librería Lombok. Lo cual nos reduce “código boilerplate”

***Lombok*** es una biblioteca de Java que se utiliza comúnmente en proyectos de Spring Boot para reducir la cantidad de código "boilerplate" necesario en las clases de entidad y otros componentes. Permite a los desarrolladores escribir código más limpio y conciso eliminando la necesidad de escribir getters, setters, constructores, y otros métodos comunes manualmente.

Clase Java Entity : La clase **Employee.java** se crea en el package **entity**.



**@Builder:** El patrón Builder es un patrón de diseño que separa la construcción de un objeto complejo de su representación, permitiendo la creación de diferentes representaciones del objeto usando un proceso de construcción paso a paso.

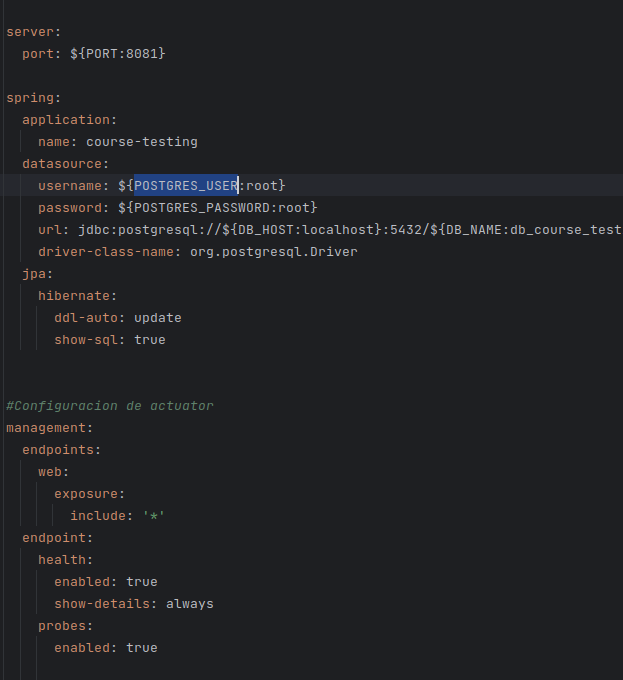
**@Id**: Nos indica que será la clave primaria

**@Column(name="first\_name")**: Esta anotación se utiliza para especificar el nombre de la columna en la base de datos a la que se mapea el campo.

2.2 Archivo de **application.properties**.

El archivo **application.properties** o **application.yaml** es un archivo de configuración clave-valor utilizado para configurar una aplicación Spring Boot de manera rápida y sencilla.

Modificamos el nombre de application.properties a **application.yaml**. en ***/src/main/resources/***



En este archivo de configuración, las variables definidas con el signo $ seguido de llaves {} son variables de entorno o variables de sistema que se utilizan para configurar dinámicamente la aplicación Spring Boot. Estas variables permiten configurar la aplicación en función del entorno en el que se está ejecutando, lo que facilita la portabilidad y la configuración sin necesidad de modificar el código fuente.

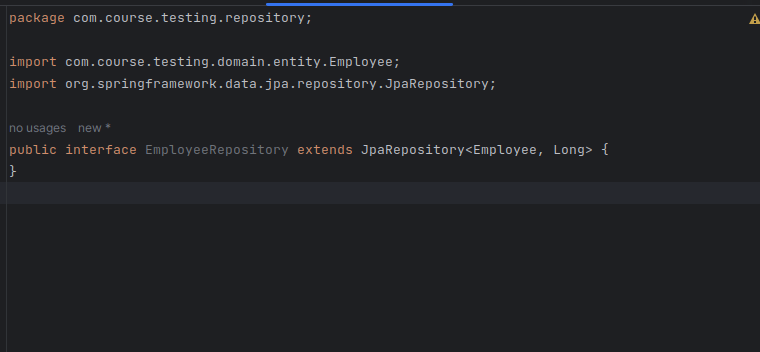
Aquí hay una explicación de cada una de las variables utilizadas:

* ${PORT:8081}: Esta variable se utiliza para configurar el puerto en el que se ejecutará el servidor. Si la variable de entorno PORT está definida, se utilizará su valor. De lo contrario, se utilizará el valor predeterminado 8081.
* ${POSTGRES\_USER:root}: Esta variable se utiliza para configurar el nombre de usuario para la base de datos PostgreSQL. Si la variable de entorno POSTGRES\_USER está definida, se utilizará su valor. De lo contrario, se utilizará el valor predeterminado root.
* ${POSTGRES\_PASSWORD:root}: Similar a la anterior, esta variable se utiliza para configurar la contraseña para la base de datos PostgreSQL. Si la variable de entorno POSTGRES\_PASSWORD está definida, se utilizará su valor. De lo contrario, se utilizará el valor predeterminado root.
* ${DB\_HOST:localhost}: Esta variable se utiliza para configurar el host de la base de datos PostgreSQL. Si la variable de entorno DB\_HOST está definida, se utilizará su valor. De lo contrario, se utilizará el valor predeterminado localhost.
* ${DB\_NAME:db\_course\_testing}: Esta variable se utiliza para configurar el nombre de la base de datos PostgreSQL. Si la variable de entorno DB\_NAME está definida, se utilizará su valor. De lo contrario, se utilizará el valor predeterminado db\_course\_testing.

Estas variables permiten que la configuración de la aplicación sea más flexible y adaptable a diferentes entornos sin necesidad de realizar cambios en el código fuente. Por ejemplo, al desplegar la aplicación en un entorno de producción, puedes configurar fácilmente las variables de entorno para adaptarse a la configuración específica de ese entorno sin modificar el archivo de configuración de la aplicación.

2.3 Creación de Repository Empleado.

A continuación creamos la interfaz Empleado Repository correspondiente a la entidad Empleado dentro del package /src/main/repository



* @Repository: Es una anotación de Spring que se utiliza para marcar una clase como un componente de acceso a datos (DAO - Data Access Object). Indica que la clase cumple con el rol de acceder a la capa de persistencia o base de datos. Por lo general, las clases anotadas con @Repository se encargan de realizar operaciones de lectura y escritura en la base de datos.
* JpaRepository: **Es una interfaz proporcionada por Spring Data JPA, que simplifica la creación de repositorios de datos basados en JPA. Esta interfaz ya proporciona métodos para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre las entidades JPA. Al extender esta interfaz en una interfaz personalizada, Spring Data JPA se encarga automáticamente de generar la implementación en tiempo de ejecución. Por lo tanto, no es necesario proporcionar la implementación de los métodos CRUD en la clase que extiende JpaRepository.**

En este caso no se utiliza @Repository en la clase ***EmployeeRepository*** debido a que a la clase SimpleJpaRepository es una clase de implementación proporcionada por Spring Data JPA que implementa la interfaz JpaRepository. Esta clase proporciona la implementación concreta de los métodos definidos en JpaRepository, como save(), findById(), findAll(), deleteById(), entre otros.

En resumen, SimpleJpaRepository es responsable de proporcionar la lógica subyacente para realizar operaciones CRUD en la base de datos para las entidades JPA. Esta implementación se basa en JPA y está diseñada para ser genérica y reutilizable, lo que permite simplificar el desarrollo de repositorios de datos en aplicaciones Java con Spring y JPA

**Nota 1:**  Todos los métodos implementados en JpaRepository son @Transactional

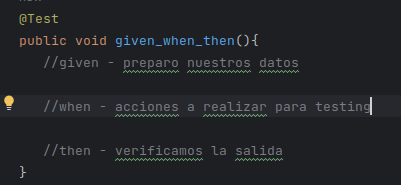
@Transactional es una anotación en Spring Framework que se utiliza para definir el ámbito de transacción para métodos o clases en una aplicación Spring. Esta anotación puede aplicarse a métodos individuales o a nivel de clase para especificar que todos los métodos en esa clase deben ejecutarse dentro de una transacción.

Al usar @Transactional, Spring maneja automáticamente la gestión de transacciones para los métodos anotados. Cuando se llama a un método anotado con @Transactional, Spring inicia una transacción antes de la ejecución del método y la finaliza después de que el método se complete. Si una excepción ocurre durante la ejecución del método, Spring realiza un rollback de la transacción para revertir cualquier cambio realizado en la base de datos.

**3 - Testing a la capa de Persistencia.**

**3.1 BDD(Behavior-Drive-Development)**

Para preparar los métodos utilizamos el patrón "Given-When-Then" es una forma de estructurar pruebas de software, especialmente pruebas de aceptación o pruebas de comportamiento, para hacerlas más legibles y comprensibles. Este patrón se utiliza comúnmente en el contexto de pruebas automatizadas y BDD (Behavior-Driven Development).



* ***Given***: Configuramos el estado inicial de la prueba, como crear objetos necesarios y configurar comportamientos simulados utilizando Mockito.
* ***When***: Ejecutamos el método que estamos probando.
* ***Then***: Verificamos los resultados esperados y que se hayan llamado los métodos esperados en los mocks

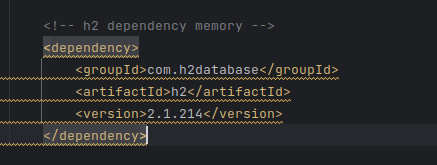
**3.2 - @DataJpaTest.**

Para realizar Testing crearemos dentro de testing crearemos el package repository, creamos la clase **EmployeeRepositoryTest**. Agregando la anotación **@DataJpaTest** .

@DataJpaTest es una anotación proporcionada por Spring Boot que se utiliza para probar componentes relacionados con JPA en una aplicación Spring Boot. Esta anotación se usa comúnmente en pruebas de integración para probar capas de persistencia de una aplicación.

Cuando se aplica @DataJpaTest a una clase de prueba, Spring Boot configura automáticamente el contexto de la aplicación para la prueba, incluyendo solo los componentes necesarios para la capa de persistencia (como repositorios, entidades y configuraciones relacionadas). Esto permite probar las operaciones de acceso a datos de la aplicación sin la necesidad de cargar todo el contexto de la aplicación, lo que puede ser más eficiente y rápido.

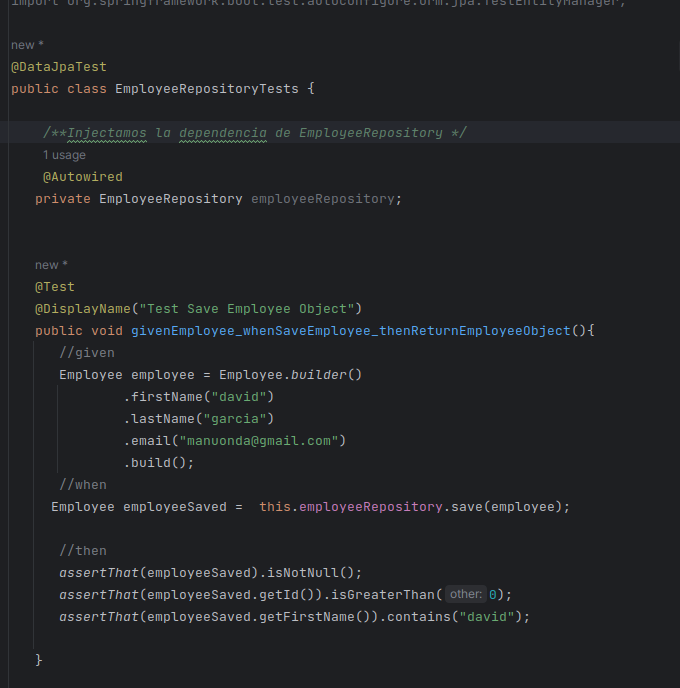
Además, @DataJpaTest **también proporciona una base de datos en memoria (como H2) para las pruebas, lo que evita la necesidad de configurar una base de datos real para las pruebas. Para esto es necesario tener la dependencia de H2. Agregada en el archivos de pom.xml**

****

Por defecto escanea classes **@Entity** y configuraciones de Spring Data JPA repository anotados con la anotaciòn **@Repository**

Por defecto todos los tests anotados con **@DataJpaTest** son transactions and roll back al final de cualquier test.

**3.3** Test Save Employee



3.4 Create Template for Junit.

Vamos a crear un template para que automaticamente al ingresar una palabra nos cargue el template base de funciones para testing . El codigo es el siguiente :

//Test for

@Test

@DisplayName("")

public void given\_when\_then(){

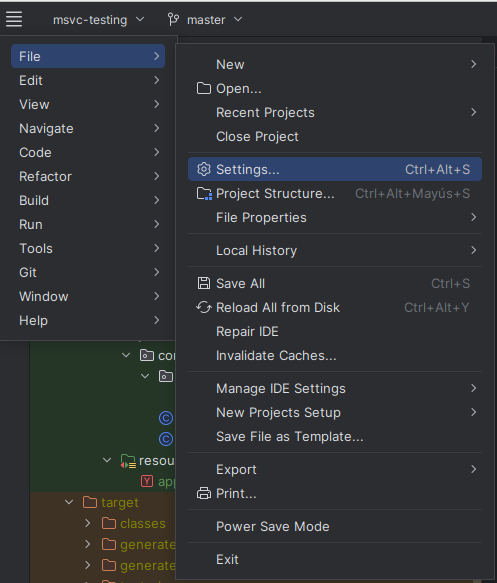
//given - preparo nuestros datos

//when - acciones a realizar para testing

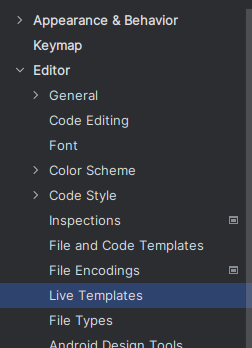
//then - verificamos la salida

}

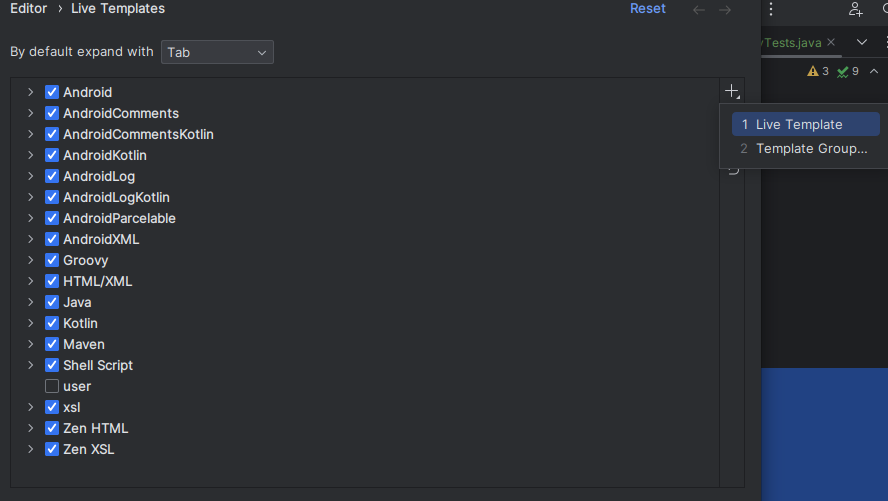
Seleccionamos del menú **File -> Settings.**



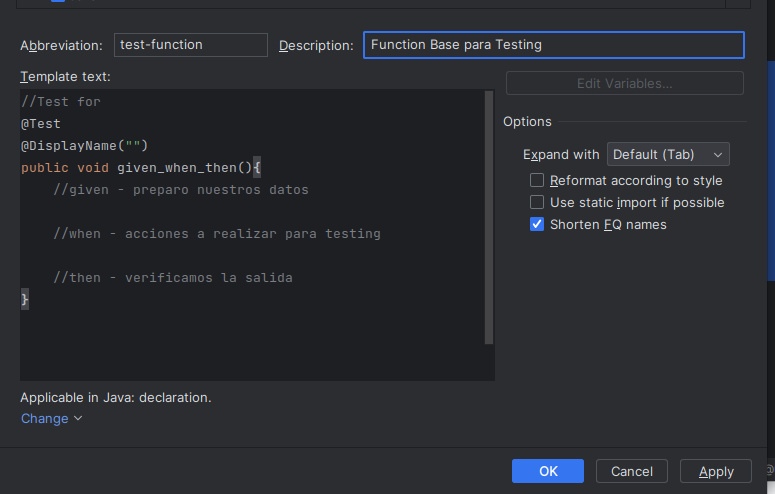
A continuaciòn seleccionamos



Despues seleccionamos Live Template



Y nos abre una opcion siguiente :

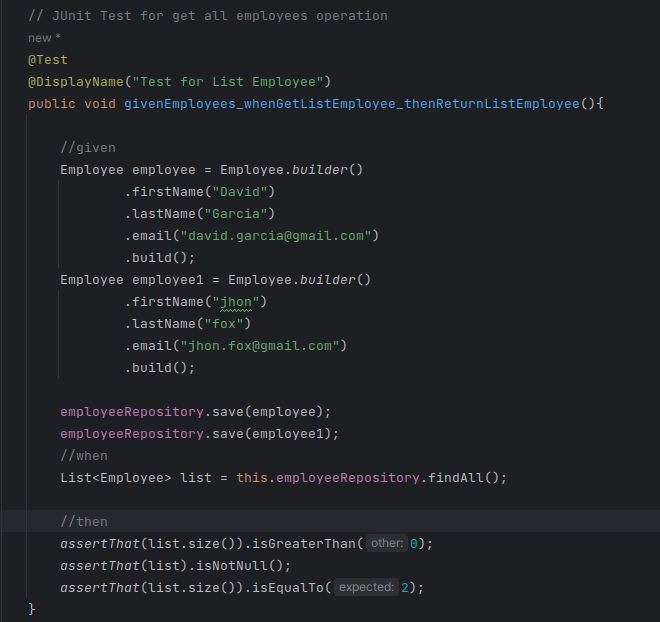


Allí completamos el campo **Abbreviation** : test-function

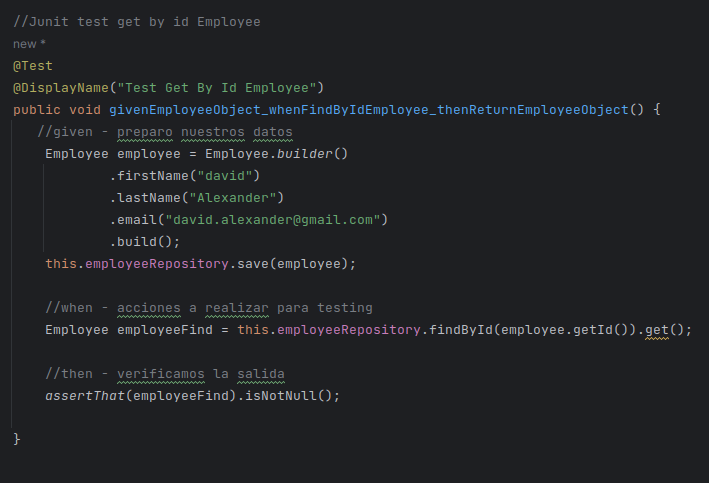
Y pegamos el código en Template Text. Seleccionamos “**Apply**” y “**OK**”.

Al ingresar en la funcion “test-function” automáticamente se completa con el template creado.

**3.5** - Test Lista de Employee

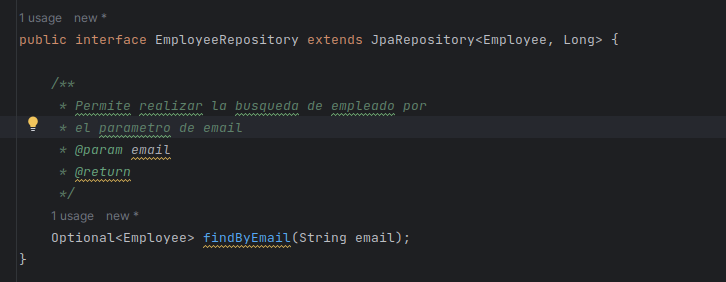


3.6 Get By Id de Employee



**3.7 Test find by email**

Como el metodo no existe en el crud repository de JpaRepository, entonces en la interfaz EmployeeRepository creamos el metodo siguiente :





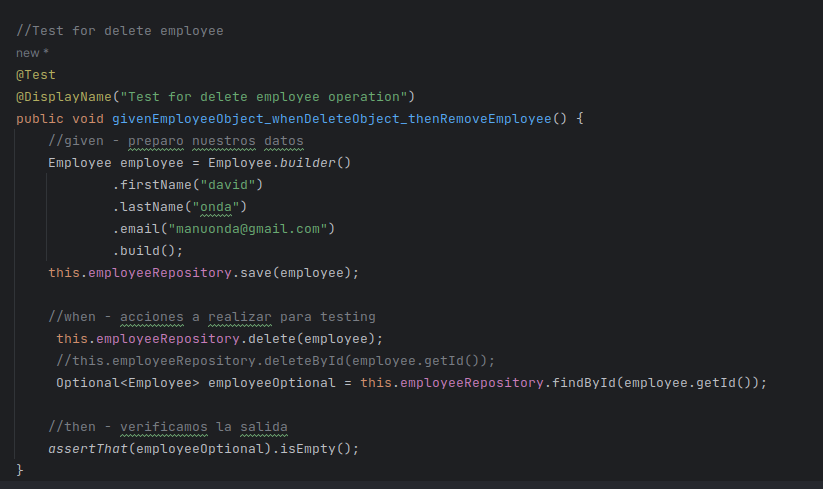
Email es un campo de la clase ***Employee.java***.

Repasando nosotros hemos creado un custom method para buscar por una propiedad especificada. Puesto que este el método no existe en las operaciones de crud de jpa repository.

**3.8** Junit Test para actualizar el Employee.



3.9 – Delete Employee

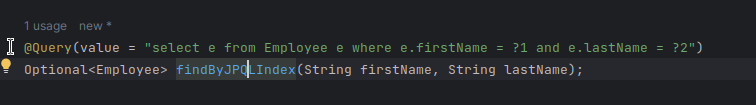


3.10 . JPLQ Java Persistence Query Language por Indexacion

JPQL, o Java Persistence Query Language, es un componente fundamental de la Java Persistence API (JPA). Es un lenguaje de consulta que permite realizar operaciones de consulta en objetos de entidad en una base de datos relacional.

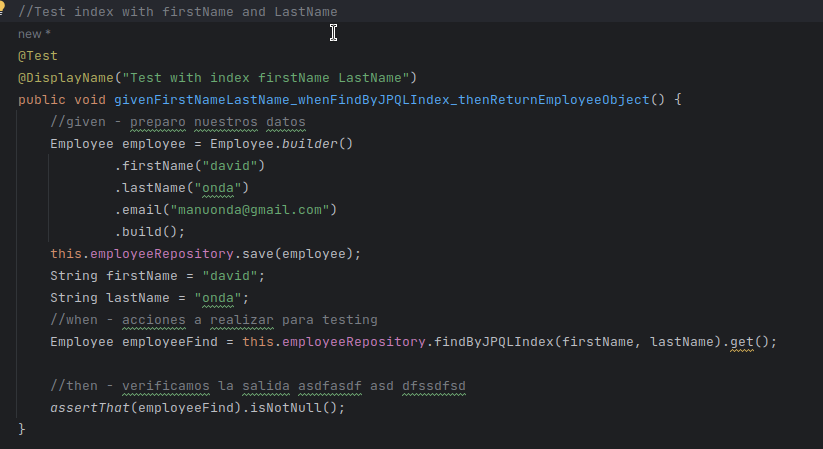
En este caso utilizaremos JPQL dentro de la interfaz EmployeeRepository crearemos

el metodo :



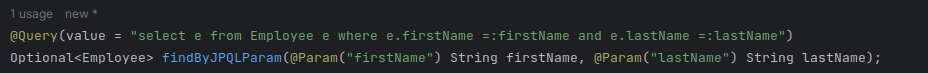
Donde los parametros ?1 y ?2 corresponde a firstName y LastName respectivamente.

El test correspondiente sería el siguiente :

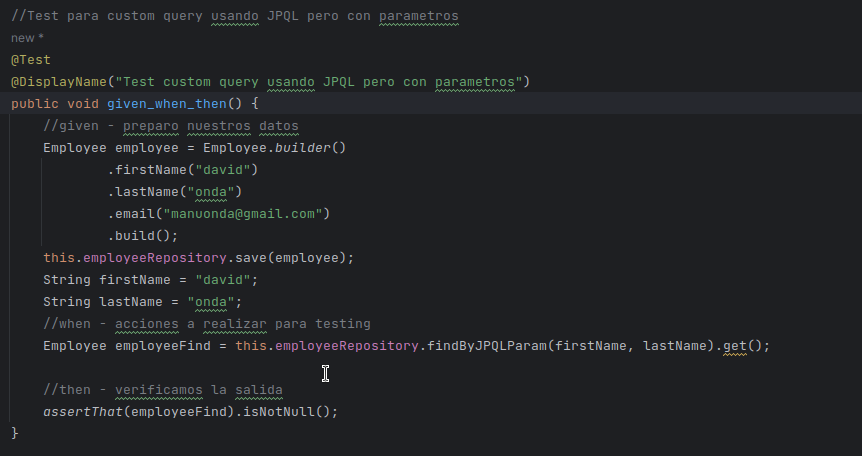


**3.11** Test para JPQL con parámetros.

En este caso usando la anotaciòn @param

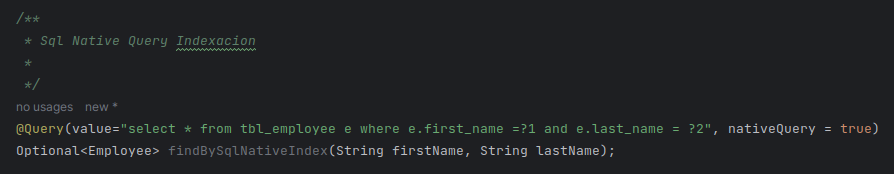


Y el test correspondiente para esta función es la siguiente :



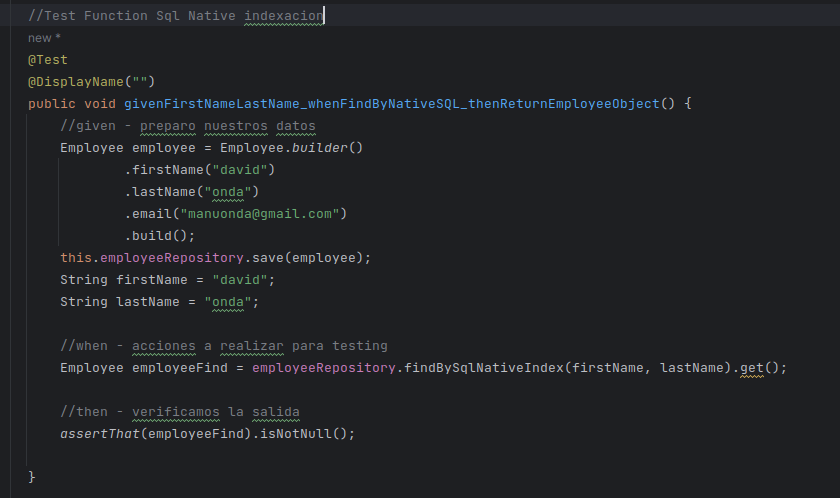
3.12 Custom Native query index.

La anotación @Query en Spring Data JPA admite la opción nativeQuery, que se utiliza para indicar si la consulta especificada es una consulta SQL nativa en lugar de una consulta JPQL. A continuaciòn se agrega el parametro **naviteQuery** = true.



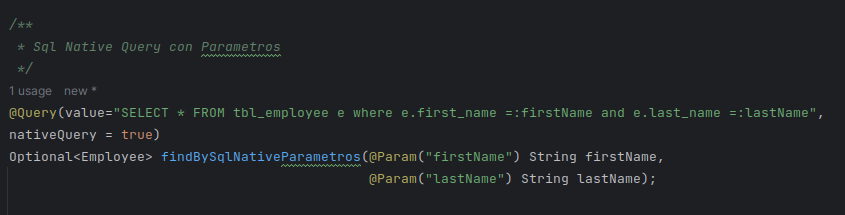
Al ser una consulta nativa en SQL , se tiene que indicar el nombre de la tabla y los nombres de columnas correspondientes y no los campos de la clase Employee.java.(Entity).

Test correspondiente.

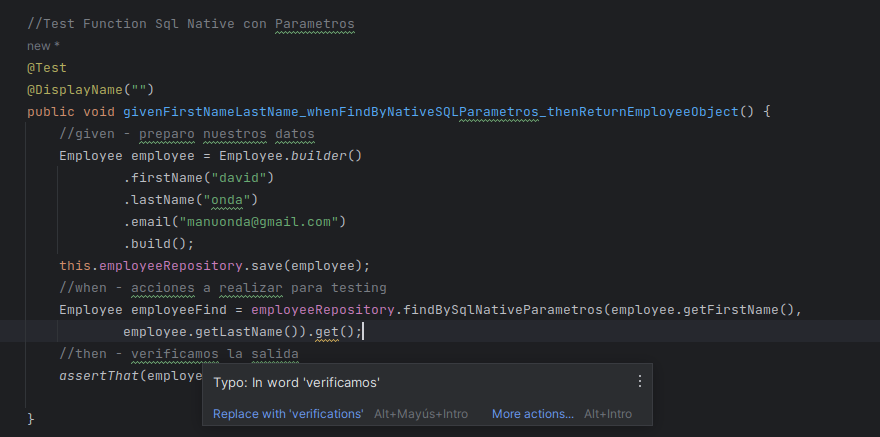


3.13 Custom Native query con parametros

En el test anterior se creó un test unitario para una consulta sql native con indexación, en este caso se realizará test unitario para sql Native pero con nombre de parámetros.



Test correspondiente al Metodo **findBySqlNativeParametros**.



3.14 Refactorizando @**Beforeach**