Prueba Técnica

Situm

Juan Antonio Martín Pichardo

Índice

1. Introducción

- **Nombre del proyecto**: API de Gestión de Edificios y Habitaciones
- **Objetivo**: Gestionar edificios y habitaciones mediante un CRUD con persistencia en base de datos.

2. Tecnologías y dependencias utilizadas

- Java 17
- Postman
- Spring Boot 3.4.3
    - Dependencias de Spring Boot
        - spring-boot-starter-data-jpa
        - spring-boot-starter-web
        - spring-boot-devtools
        - spring-boot-starter-test
        - Base de datos H2
- Maven
- Control de versiones Github

3. Estructura del proyecto

Para estructurar el proyecto, he decidido utilizar el modelo MVC (Modelo-Vista-Controlador), ya que es una arquitectura simple, escalable y ampliamente utilizada en el desarrollo de aplicaciones.

Las clases se organizan en paquetes para mantener una estructura clara y modular. En este caso, el paquete principal de la aplicación es **com.situm.situm**, actuando como el paquete padre. Todos los demás deben estar organizados dentro de este para garantizar una correcta jerarquía y organización del código. Esta es la estructura del proyecto:

- **src/main/java**
    - **com.situm.situm**
        - **SitumApplication**. Esta es la clase principal del proyecto y la que se usa para poder ejecutar la aplicación
    - **com.situm.situm.Controller**
        - **EdificioController:** Esta clase se encarga de gestionar las peticiones HTTP relacionadas con la entidad **Edificio**. En ella se implementan

métodos que permiten obtener, actualizar, añadir y eliminar registros de la tabla **Edificio**, según la operación que se requiera.

Las acciones pueden realizarse mediante parámetros en las solicitudes, permitiendo, por ejemplo, buscar o eliminar un edificio con su **ID** como referencia.

- **HabitacionController**. Esta clase se encarga de gestionar las peticiones HTTP relacionadas con la entidad **Habitación.** En ella se implementan métodos que permiten obtener, actualizar, añadir y eliminar registros de la tabla **Habitación**, según la operación que se requiera.

- **com.situm.situm.Enums:** Al utilizar un enum, se previenen errores y se garantiza la consistencia de la información, asegurando que solo se registren valores válidos y estandarizados.

  - **Categoría:** Este enumerador define las diferentes categorías en las que pueden clasificarse las habitaciones. Su uso permite un mayor control sobre los datos almacenados en la base de datos, restringiéndolos a valores predefinidos.

  - **Estado:** Este enumerador define los distintos tipos de estados en los que pueden encontrarse las habitaciones. Su uso garantiza un mayor control sobre los datos ingresados en la base de datos, ya que restringe los valores a opciones predefinidas.

  - **TipoVia:** Este enumerador define los distintos tipos de vía en los que pueden encontrarse los edificios. Su uso garantiza un mayor control sobre los datos ingresados en la base de datos, ya que restringe los valores a opciones predefinidas.

- **com.situm.situm.Model:**

  - **EdificioModel:** Esta clase representa la entidad **Edificio** y se encarga de mapear sus atributos con la tabla correspondiente en la base de datos. Define las propiedades del edificio y su relación con otras entidades, permitiendo la persistencia y gestión de datos mediante **JPA (Java Persistence API)**.

  - **HabitacionModel** Esta clase representa la entidad **Habitación** y se encarga de mapear sus atributos con la tabla correspondiente en la base de datos. Define las propiedades del edificio y su relación con otras entidades, permitiendo la persistencia y gestión de datos mediante **JPA (Java Persistence API)**.

- **com.situm.situm.Repository:**

  - **edificioRepository:** Esta interfaz extiende de JpaRepository y proporciona los métodos necesarios para realizar operaciones sobre la tabla Edificios en la base de datos.

    Al ser utilizada en el *service*, hereda automáticamente un conjunto de métodos predeterminados de JpaRepository, como guardar, eliminar y

buscar registros. Además, es posible definir métodos personalizados, por ejemplo: ***public EdificioModel buscarPorPais(String pais);***

- **HabitacionRepository:** Esta interfaz se extiende de JpaRepository y proporciona los métodos necesarios para realizar operaciones sobre la tabla Habitaciones en la base de datos.

  Al ser utilizada en el *service*, hereda automáticamente un conjunto de métodos predeterminados de JpaRepository, como guardar, eliminar y buscar registros. Además, es posible definir métodos personalizados, por ejemplo: ***public HabitacionModel buscarPorLetra(String letra);***

- **com.situm.situm.Service:**

  - **EdificioService:** La función principal de esta clase es gestionar las operaciones relacionadas con la entidad **EdificioModel**, realizando llamadas al repositorio correspondiente. Estas operaciones permiten manipular los datos de los edificios y posteriormente ser utilizadas en las peticiones HTTP.

  - **HabitacionService:** La función principal de esta clase es gestionar las operaciones relacionadas con la entidad **HabitaciónModel**, realizando llamadas al repositorio correspondiente. Estas operaciones permiten manipular los datos de las habitaciones y posteriormente ser utilizadas en las peticiones HTTP.

- **src/main/resources:**

  - **application.properties:** En este archivo se gestionan las propiedades de la aplicación, como la configuración de la base de datos, el modo *debug* y las credenciales de acceso a la base de datos, entre otros parámetros.

4. Modelo entidad-relación

En este apartado se describe el modelo entidad-relación de las tablas de la base de datos, el tipo de datos utilizado para cada atributo y la lista completa de las tablas existentes.

Para esta prueba se utilizan dos tablas: la tabla '**Edificios**' y la tabla '**Habitaciones**', las cuales están basadas en un modelo de relación 1 a N. Esto significa que un edificio puede tener muchas habitaciones, pero una habitación solo puede pertenecer a un único edificio.

Para poder establecer la relación primero se debe haber creado la tabla '**Edificios**' la cual es la tabla principal. Esta contiene los siguientes atributos:

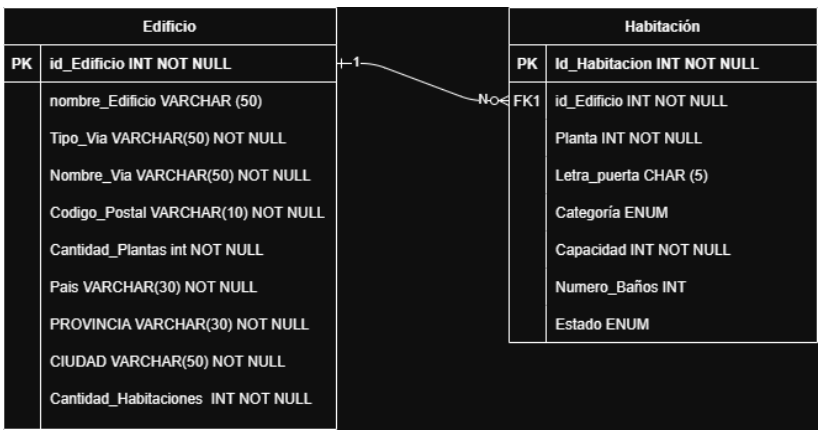
- **idEdificio**: Es un int que se auto-incrementa cuando se añade un nuevo registro. Este atributo funciona como la clave primaria de esta tabla. Este valor es único y no se puede repetir
- **NombreEdificio**: Es un varchar(50) que nos daría el nombre del edificio
- **TipoVia**: Es un varchar(50) que nos daría el tipo de vía en el que se encuentra el edificio

- **NombreVia**: Es un varchar(50) que nos daría el nombre de la vía en la que se encuentra el edificio
- **CopdigoPostal**: Es un varchar(10) que nos daría el código postal en el que se encuentra el edificio. La decisión de que sea de tipo varchar en vez de int es debido a que hay países en las que el código postal puedo incluir una letra o caracteres especiales
- **CantidadPlantas**: Este atributo sería un int que nos da la cantidad de plantas que contiene un edificio
- **Pais**: Este atributo sería un varchar(50) que nos da el nombre del país en el que se encuentran el edificio
- **Provincia**: Es un varchar(50) que nos da el nombre de la provincia en la que se encuentra el edificio
- **Ciudad**: Es un varchar(50) que nos da el nombre de la ciudad en la que se encuentra el edificio
- **CantidadHabitaciones**: Es un int que nos da la cantidad total de habitaciones que puede tener un edificio

Una vez se ha creado la tabla **edificios** podemos crear la tabla **Habitaciones**:

- **IdHabitacion**: Es un int que se auto-incrementa cuando se añade un nuevo registro. Este atributo funciona como la clave primaria de esta tabla, es único y no se puede repetir.
- **IdEdificio**: Es un int que actúa como clave foránea, lo que significa que está relacionado con la clave primaria de la tabla 'Edificios'. Esto se utiliza para mantener la relación entre ambas tablas y garantizar que estén conectadas.
- **Planta**: Es un int que indica el número de planta en el que se encuentra una habitación.
- **letraPuerta**: Es un char al que solo se le puede añadir una letra, este nos indica la letra que tiene la habitación. Esto permite identificar la habitación.
- **Categoría**: Es un varchar(30) que nos indica la categoría a la que pertenece la habitación**.**
- **Capacidad**: Es un int que da la cantidad de personas que pueden estar en esta habitación.
- **NumeroBaños**: Es un int que da la cantidad de baños que puede tener la habitación.
- **Estado**: Es un varchar(30) que nos indica el estado en el que se encuentra la habitación.

Una vez creadas ambas tablas se pueden insertar datos en ellas. Este sería el esquema de la entidad-relación de las tablas:



***Esta vista no es la reflejada en la base de datos. Este es solo un modelo que se ha usado para poder tener una referencia**

5. Endpoint de la API

Los endpoints de la aplicación son rutas que permiten realizar diversas operaciones, como la obtención de datos, su modificación o el envío de información. Estos endpoints están definidos en las clases de controlador: **'EdificioController'** y **'HabitacionController'**.

En dichos endpoints se utilizan anotaciones que ayudan a identificar qué acción se realizará con los datos. Aunque algunas anotaciones pueden repetirse, esto se debe a que una misma anotación puede aplicarse a varios métodos, solo cambiando los parámetros de la ruta para hacer una búsqueda más específica. Por ejemplo, se puede usar `@GetMapping` para el método que obtiene todos los registros y la misma anotación, añadiendo `("/{id}")` al final, para indicar que después de la ruta debe especificarse un identificador.

Un ejemplo práctico sería la URL: `http://localhost:8080/api/habitaciones/4`, donde el número 4 se utiliza para buscar la habitación con ese ID.

Estos son los EndPoint usados en la aplicación:

- **Endpoint de Edificios:**
    - **/api/edificios [GET]:** Devuelve todos los datos de los edificios.
    - **/api/edificios [POST]:** Se utiliza para crear nuevos registros de edificios.
    - **/api/edificio/{id} [PUT]:** Se utiliza para modificar y actualizar los datos de un edificio específico por su ID.
    - **/api/edificios/{id} [DELETE]:** Permite eliminar un edificio según su ID.
    - **/api/edificios/{id} [GET]:** Permite buscar los detalles de un edificio específico por su ID.
- **Endpoint de Habitaciones:**
    - **/api/habitaciones [GET]:** Devuelve todos los datos de las habitaciones.
    - **/api/habitaciones [POST]:** Se utiliza para crear nuevos registros de habitaciones.
    - **/api/habitaciones/{id} [PUT]:** Se utiliza para modificar y actualizar los datos de una habitación específica por su ID.
    - **/api/habitaciones/{id} [DELETE]:** Permite eliminar una habitación según su ID.
    - **/api/habitaciones/{id} [GET]:** Permite buscar los detalles de una habitación específica por su ID.

Estos son los endpoints básicos para una aplicación. Sin embargo, es posible crear más endpoints según las necesidades. Para ello, será necesario crear los métodos correspondientes en las interfaces y servicios correspondientes, de modo que puedan ser utilizados en los controladores

6. Configuración de la base de datos

Para poder hacer la configuración de la base de datos debemos añadir ciertas características en el application.properties:

```
spring.application.name=situm
spring.datasource.url=jdbc:h2:file:./data/testdb;DB_CLOSE_ON_EXIT=FALSE;AUTO
_RECONNECT=TRUE
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=situm
spring.datasource.password=technicalTest
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
spring.main.allow-circular-references=true
```

Debido a que H2 es una base de datos utilizada para pruebas y desarrollo local, los datos se almacenan en memoria, lo que significa que no hay persistencia entre ejecuciones. Para solucionar este problema, debemos indicar en la URL:
jdbc:h2:file:./data/testdb;DB_CLOSE_ON_EXIT=FALSE;AUTO_RECONNECT=TRUE
para que la base de datos se guarde en nuestro repositorio, lo que permitirá mantener la persistencia entre ejecuciones.

Dado que H2 es una base de datos que se puede utilizar directamente al crear un proyecto Spring Boot, no es necesario tener un gestor de bases de datos instalado por separado. Solo tendríamos que incluir la siguiente instrucción en nuestro **pom.xml**.

```
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <scope>runtime</scope>
        </dependency>
```

7. Extras

En esta sección se comentan mejoras o funciones que no se han podido añadir por falta de tiempo

a. Nuevas tablas en base de datos

Una de las mejoras que se podría implementar sería la creación de una nueva tabla llamada '**Planta**'. Esta tabla tendría una relación 1 a N con la tabla '**Edificios'**, donde un edificio puede tener muchas plantas, pero una planta solo puede pertenecer a un edificio. Además, la tabla '**Planta**' estaría relacionada con la tabla '**Habitación**' en una relación 1 a N, donde una planta puede tener muchas habitaciones, pero una habitación solo puede estar en una planta.

b. Código de pruebas

Para poder realizar las pruebas de integración y unitarias primero debemos tener en nuestro pom.xml las dependencias necesarias para realizar los test. Por ejemplo:

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

*En caso de utilizar JUnit o Mockito deben añadirse las dependencias correspondientes al pom.

Una vez hecho esto, lo más eficiente sería crear un nuevo paquete dentro del paquete principal, como `com.situm.situm.test`. En él se crearían las clases correspondientes, por ejemplo, `TestEdificioService`, a la que deberíamos agregar la anotación `@SpringBootTest`.

Dentro de esta clase, podemos definir los métodos necesarios para realizar las pruebas pertinentes: por ejemplo, para una prueba unitaria, podríamos crear un método que, mediante un bucle `foreach,` llame a la función del repositorio que busca por ID, y al mismo tiempo lo vaya incrementando. De esta forma, se nos irían devolviendo los datos en función de cada ID. En este caso, estamos describiendo una prueba unitaria, ya que solo verifica que el método funciona correctamente.

En el caso de las pruebas de integración, deberíamos crear una nueva clase dentro del paquete de test, por ejemplo, `TestIntegracionEdificios`. En ella, incluiríamos una serie de métodos que comprobarán si las distintas partes de la aplicación funcionan correctamente, como un método que devuelva un código de estado 200 o 'OK' al realizar una búsqueda de todos los datos.

También podríamos tener una prueba en la que busquemos un ID que no existe o un ID erróneo para ver qué resultado nos devuelve la aplicación.

Para ejecutar estas pruebas, y dado que este proyecto está construido con Maven, debemos usar el comando `mvn test` o, si se está utilizando un IDE con integración Maven, emplear las herramientas del IDE. Por ejemplo, en VS Code, que tiene una vista de Maven, podemos hacer clic derecho sobre el proyecto y seleccionar la opción 'Iniciar Test'.

c. Sentencias usadas en base de datos

Aunque la base de datos tiene persistencia, esto no quita que pueda tener errores debido a que es una base de datos para desarrollo local y pruebas. Por ello en esta documentación se incluyen las sentencias para crear las tablas y añadir registros

Creación de la tabla Edificio y sus atributos

CREATE TABLE EDIFICIO(IDEDIFICIO INT AUTO_INCREMENT PRIMARY KEY, TIPOVIA VARCHAR(50) NOT NULL, NOMBREVIA VARCHAR(50) NOT NULL, CODIGOPOSTAL VARCHAR(10) NOT NULL, CANTIDADPLANTAS INT NOT NULL, PAIS VARCHAR(30) NOT NULL, PROVINCIA VARCHAR(30) NOT NULL, CIUDAD VARCHAR(50) NOT NULL, NHABITACION INT, NOMBREEDIFICIO VARCHAR(100))

Creación de la tabla habitaciones con sus atributos
CREATE TABLE habitacion (idhabitacion INT AUTO_INCREMENT PRIMARY KEY, idEdificio INT NOT NULL, FOREIGN KEY (idEdificio) REFERENCES edificio(idEdificio), planta INT NOT

NULL, letraPuerta CHAR(5), categoria VARCHAR(30), capacidad INT NOT NULL, NbañosHabitacion INT, estado VARCHAR(30));

Inserción de datos en la tabla edificios

INSERT INTO EDIFICIO (TIPOVIA, NOMBREVIA, CODIGOPOSTAL, CANTIDADPLANTAS, PAIS, PROVINCIA, CIUDAD, NHABITACION, nombreEdificio) VALUES
('Calle', 'Gran Vía', '28013', 10, 'España', 'Madrid', 'Madrid', 100, 'Torre Madrid'),
('Avenida', 'Diagonal', '08019', 15, 'España', 'Barcelona', 'Barcelona', 200, 'Residencial Diagonal'),
('Calle', 'Serrano', '28006', 8, 'España', 'Madrid', 'Madrid', 80, 'Edificio Serrano'),
('Boulevard', 'Haussmann', '75009', 12, 'Francia', 'Île-de-France', 'París', 150, 'Haussmann Palace'),
('Calle', 'Florida', 'C1005', 20, 'Argentina', 'Buenos Aires', 'Buenos Aires', 300, 'Torre Florida'),
('Avenida', 'Reforma', '06500', 25, 'México', 'Ciudad de México', 'Ciudad de México', 500, 'Torre Reforma'),
('Calle', 'Broadway', '10036', 30, 'Estados Unidos', 'Nueva York', 'Nueva York', 600, 'Broadway Tower'),
('Avenida', 'Paulista', '01311-000', 22, 'Brasil', 'São Paulo', 'São Paulo', 400, 'Paulista Center'),
('Calle', 'Nathan Road', '000000', 18, 'China', 'Hong Kong', 'Hong Kong', 350, 'Nathan Plaza'),
('Avenida', 'Nevsky Prospekt', '191186', 10, 'Rusia', 'San Petersburgo', 'San Petersburgo', 250, 'Nevsky Tower');

Inserción de datos en la tabla habitaciones

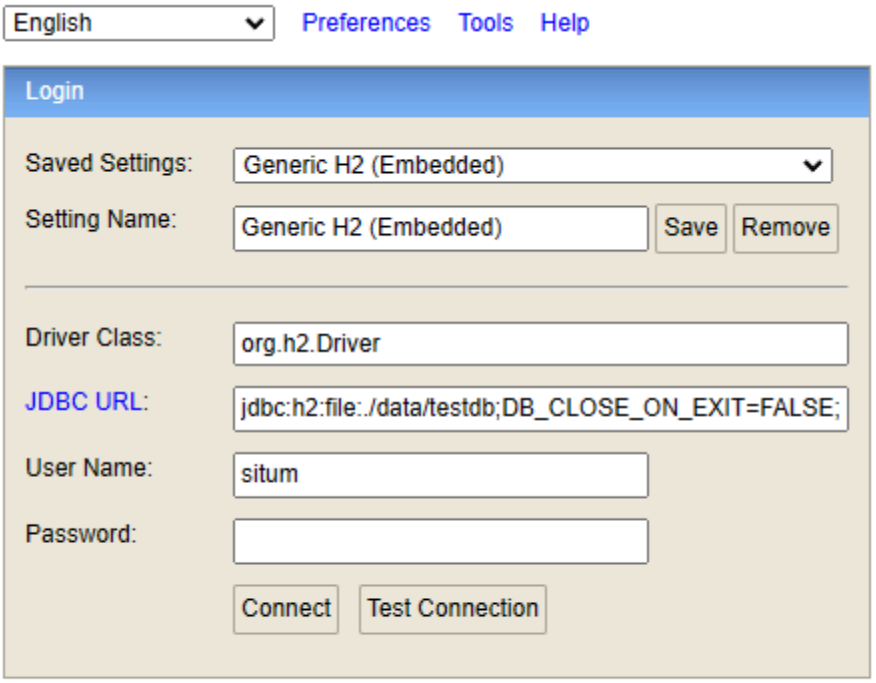
INSERT INTO habitacion (idEdificio, planta, letraPuerta, categoria, capacidad, NbañosHabitacion, estado) VALUES
(1, 1, 'A', 'Estándar', 2, 1, 'Disponible'),
(1, 2, 'B', 'Suite', 4, 2, 'Ocupado'),
(2, 3, 'C', 'Doble', 2, 1, 'Mantenimiento'),
(2, 5, 'D', 'Estándar', 1, 1, 'Disponible'),
(3, 2, 'E', 'Suite', 3, 2, 'Ocupado'),
(4, 6, 'F', 'Doble', 2, 1, 'Disponible'),
(5, 8, 'G', 'Penthouse', 5, 3, 'Reservado'),
(6, 10, 'H', 'Suite', 3, 2, 'Ocupado'),
(7, 12, 'I', 'Estándar', 2, 1, 'Disponible'),
(8, 15, 'J', 'Doble', 2, 1, 'Mantenimiento');

Estos son los datos con los que he ejecutado la aplicación y he realizado las pruebas

d. Usar la base de datos en el navegador

Para poder usar la base de datos en el navegador tenemos que usar la siguiente url https://localhost:8080/h2-console en el navegador que prefiramos. Previamente debemos tener configurada la base de datos en el application.properties la configuración previamente mostrada en esta documentación: también para poder acceder a url la aplicación debe estar en ejecución.

Una vez se tiene en cuenta todo lo anterior cuando accedemos a la url nos debería mostrar una vista como esta:



En el apartado **JDBC URI** debemos poner la url configurada en el *application.properties*. Con esto podemos usar el usuario y la contraseña que hemos definido en este fichero. Es importante saber que cada vez que se reinicie la aplicación se nos cierra la sesión de la base de datos, pero debido a la configuración que hemos realizado no perderemos los datos guardados