

Acceso a datos

None

Begoña Paterna

None

Índice

1. Presentación	3
2. Contenido	4
2.1 Unidad 2. Acceso a Bases de Datos relacionales	4
2.2 Unidad 3. Acceso a Bases de Datos documentales	27
2.3 Unidad 4. Componentes (Spring Framework)	51
3. Material adicional	104
3.1 Ubuntu Server en AWS Learner Lab	104
3.2 DB Browser for SQLite	122
3.3 DBeaver	126
3.4 MySQL	133
3.5 Instalación y administración de MongoDB	141

1. Presentación

Descripción

Estos materiales pertenecen al módulo de **acceso a datos** del ciclo superior de **desarrollo de aplicaciones multiplataforma** y están diseñados para que aprendas haciendo. A lo largo de las distintas unidades, no solo veremos la teoría, sino que la aplicaremos directamente para construir, paso a paso, una aplicación completa de gestión de datos. El tema de la aplicación lo eliges tú, pero los pasos que daremos serán los mismos para todos.

Siguiendo la unidad no solo habrás aprendido los conceptos, sino que tendrás una aplicación completa y funcional creada por ti.

Intercaladas con la teoría y con los ejemplos encontrarás tres tipos de cajas. A continuación se explica que es cada una de ellas:

Ejecutar y analizar

Estas cajas son para analizar y comprender en detalle el ejemplo de código proporcionado. Tu tarea es ejecutar ese código, observar la salida y asegurarte de entender cómo y por qué funciona.

Práctica: Aplicar y construir

Estas cajas son prácticas que debes realizar tú. Es el momento de ponerte a programar y aplicar lo que acabas de aprender. Son los objetivos que debes completar para avanzar. Cada una de estas prácticas es un bloque que debes programar para ir avanzando en tu proyecto final. En cada práctica ampliarás lo de las anteriores.

Entrega

Estas cajas son entregas de tu trabajo. Las entregas pueden ser parciales (la profesora te dará sugerencias de mejora) o finales (la profesora calificará el trabajo que has realizado). No todas las prácticas llevan asociada una entrega.

Autoría

Obra realizada por Begoña Paterna Lluch. Publicada bajo licencia [Creative Commons Atribución/Reconocimiento-CompartirIgual 4.0 Internacional](#)

2. Contenido

2.1 Unidad 2. Acceso a Bases de Datos relacionales

Revisiones

Revisión	Fecha	Descripción
1.0	05-10-2025	Adaptación de los materiales a markdown
1.1	16-10-2025	Ampliación de ejemplos y prácticas
1.2	21-10-2025	Inclusión del punto 6 (funciones y procedimientos almacenados)
1.3	26-10-2025	Modificación del Ejemplo 5: commit y rollback

2.1.1 2.1.1. Introducción

Las bases de datos relacionales son esenciales en el desarrollo de aplicaciones modernas. Su integración con una aplicación requiere realizar una **conexión** al sistema gestor de base de datos (SGBD) desde el lenguaje de programación. Este tema se centra en cómo realizar esa conexión, cómo trabajar con datos mediante sentencias SQL y cómo aplicar buenas prácticas, como el cierre de recursos, el uso de transacciones y procedimientos almacenados.

Una **base de datos relacional** es un sistema de almacenamiento de información que **organiza los datos en tablas**. Cada tabla representa una entidad (por ejemplo, clientes, productos, facturas) y está compuesta por filas y columnas, donde cada fila representa un registro único y cada columna contiene un atributo específico de ese registro. Estas bases de datos (BD) siguen el **Modelo Relacional**, desarrollado por Edgar F. Codd en la década de 1970, y permite establecer vínculos o **relaciones entre diferentes tablas** mediante **claves primarias y foráneas**, facilitando así la integridad, la coherencia y la eficiencia en el manejo de grandes volúmenes de datos.

Ejemplo de tabla `clientes`:

id_cliente	nombre	ciudad
1	Pol	Castellón
2	Eli	Valencia

La **Clave primaria (Primary Key)** es una columna (o conjunto de columnas) que **identifica de forma única** cada fila de una tabla. En el ejemplo: `id_cliente` es clave primaria en la tabla `clientes`.

La **Clave foránea (Foreign Key)** es una columna que **hace referencia a una clave primaria de otra tabla** para establecer una relación. Por ejemplo, en otra tabla llamada `facturas`, el campo `id_cliente` puede ser clave foránea que apunta al campo `id_cliente` de la tabla `clientes`.

Ejemplo de tabla `facturas`:

id_factura	id_cliente	fecha
1	1	2025-09-18
2	1	2025-09-18

El lenguaje **SQL (Structured Query Language)** se utiliza para gestionar bases de datos relacionales ya que gracias a él se pueden crear, modificar, consultar y eliminar datos de forma sencilla y estandarizada. Es lo que se denomina **CRUD**, es decir, Create (crear), Read (Leer), Update (Actualizar) y Delete (Borrar). Esto lo convierte en la opción preferida para una amplia variedad de aplicaciones empresariales y tecnológicas.

Algunos de sus comandos básicos son:

- SELECT : consultar datos
- INSERT : añadir registros
- UPDATE : modificar datos existentes
- DELETE : eliminar registros
- CREATE : definir tablas, claves, relaciones, etc.

Un ejemplo sencillo de consulta podría ser:

```
SELECT nombre FROM clientes WHERE ciudad = 'Valencia';
```

Tipos de SGBD relacionales

Conocer qué **tipo de gestor de base de datos** se está utilizando es esencial para poder **conectar** correctamente desde una aplicación, ya que cada uno necesita su propio conector o driver. Podemos encontrar:

1. Gestores independientes (cliente-servidor): PostgreSQL, MySQL, Oracle, SQL Server...

- Sistemas robustos y escalables, ideales para entornos multi-usuario y aplicaciones web.
- Requieren un servidor dedicado y una configuración más compleja.
- Casos de uso: aplicaciones web, servicios empresariales, sistemas con alta demanda de acceso concurrente.

2. Gestores embebidos: SQLite, H2, Derby...

- Base de datos ligera, sin servidor, ideal para aplicaciones móviles o de escritorio donde no se requiere gestión centralizada.
- Fácil de configurar y desplegar, ya que la base de datos reside en un archivo local.
- Casos de uso: aplicaciones de escritorio, móviles, prototipos, pruebas unitarias.

Práctica 1: Crea tu base de datos

A partir del fichero de información utilizado en el proyecto de la unidad anterior, crea una base de datos SQLite **nombre_de_tu_BD.sqlite** con una tabla que contenga la información del fichero. Puedes utilizar [DB Browser for SQLite](#)

2.1.2 2.2. Conexión a un SGBD

Cuando desarrollamos aplicaciones que trabajan con información persistente, necesitamos acceder a BD para consultar, insertar, modificar o eliminar datos. Existen dos formas principales de hacerlo desde el código:

- Acceso mediante ORM (Object-Relational Mapping).
- Acceso mediante conectores.

Acceso mediante ORM

Un **ORM** es una herramienta que permite trabajar con la base de datos como si fuera un conjunto de objetos, evitando tener que escribir directamente SQL. El **ORM** se encarga de mapear las tablas a clases y los registros a objetos, y traduce automáticamente las operaciones del código a consultas SQL. Es ideal para trabajar de forma más productiva en aplicaciones complejas. Sus principales características son:

- Se trabaja con clases en lugar de tablas SQL.
- Ahorra mucho código repetitivo.
- Ideal para proyectos medianos o grandes que requieren mantener muchas entidades.

Algunos ejemplos de ORMs

ORM / Framework	Lenguaje	Descripción
Hibernate	Java/Kotlin	El ORM más utilizado con JPA
Exposed	Kotlin	ORM ligero y expresivo creado por JetBrains
Spring Data JPA	Java/Kotlin	Abstracción que automatiza el acceso a datos
Room	Java/Kotlin	ORM oficial para bases de datos SQLite en Android

JPA (Java Persistence API) es una especificación estándar de Java que define cómo se deben mapear objetos Java (o Kotlin) a tablas de bases de datos relacionales. Es decir, permite gestionar la persistencia de datos de forma orientada a objetos, sin necesidad de escribir SQL directamente. Es el estándar utilizado por las herramientas ORM como Hibernate, EclipseLink, o Spring Data JPA.

Acceso mediante conectores

Un **conector** (también llamado driver) es una librería software que permite que una aplicación se comunique con un gestor de base de datos (SGBD). Actúa como un puente entre nuestro código y la base de datos, traduciendo las instrucciones SQL a un lenguaje que el gestor puede entender y viceversa. Sin un conector, tu aplicación no podría comunicarse con la base de datos.

Una base de datos puede ser accedida desde diferentes orígenes o herramientas, siempre que tengamos:

- Las credenciales de acceso (usuario y contraseña)
- El host/servidor donde se encuentra la base de datos
- El motor de base de datos (PostgreSQL, MySQL, SQLite, etc.)
- Los puertos habilitados y los permisos correctos

Las principales formas de conectarse a una base de datos son las siguientes:

Medio de conexión	Descripción
Aplicaciones de escritorio	Herramientas gráficas como DBeaver , pgAdmin , MySQL Workbench , DB Browser for SQLite . Permiten explorar, consultar y administrar BD de forma visual.
Aplicaciones desarrolladas en código	Programas en Kotlin , Java , Python , C# , etc., mediante conectores como JDBC , psycopg2 , ODBC , etc. para acceder a BD desde código.
Línea de comandos	Clients como <code>psql</code> (PostgreSQL), <code>mysql</code> , <code>sqlite3</code> . Permiten ejecutar comandos SQL directamente desde terminal.
Aplicaciones web	Sitios web que acceden a BD desde el backend (por ejemplo, en Spring Boot, Node.js, Django, etc.).
APIs REST o servidores intermedios	Servicios web que conectan la BD con otras aplicaciones, actuando como puente o capa de seguridad.
Aplicaciones móviles	Apps Android/iOS que acceden a BD locales (como SQLite) o remotas (vía Firebase , API REST, etc.).
Herramientas de integración de datos	Software como Talend , Pentaho , Apache Nifi para migrar, transformar o sincronizar datos entre sistemas.

De todas las formas posibles de interactuar con una base de datos, nos vamos a centrar en el uso de **conectores JDBC (Java Database Connectivity)**. Una aplicación (escrita en Kotlin, Java u otro lenguaje) puede leer, insertar o modificar información almacenada en una base de datos relacional si previamente se ha conectado al sistema gestor de base de datos (SGBD). **JDBC** es una API estándar de Java (y compatible con Kotlin) que permite conectarse a una BD, enviar instrucciones SQL y procesar los resultados manualmente. Es el método de más bajo nivel, pero ofrece un control total sobre lo que ocurre en la BD. Es ideal para aprender los fundamentos del acceso a datos y aprenderlo ayuda a entender mejor lo que hace un ORM por debajo.

Sus principales características son:

- El programador escribe directamente las consultas SQL.
- Requiere gestionar manualmente conexiones, sentencias y resultados.
- Se necesita un driver específico (conector) para cada SGBD:

A continuación se muestra su sintaxis general. Aunque puede variar según el SGBD con el que se trabaje. Por ejemplo en SQLite no se necesita usuario ni contraseña ya que es una base de datos local y embebida:

```
jdbc:<gestor>://<host>:<puerto>/<nombre_base_datos>
```

Algunos ejemplos de conectores según el SGBD

SGBD	Conector (Driver JDBC)	URL de conexión típica
PostgreSQL	org.postgresql.Driver	jdbc:postgresql://host:puerto/nombreBD
MySQL / MariaDB	com.mysql.cj.jdbc.Driver	jdbc:mysql://host:puerto/nombreBD
SQLite (embebido)	org.sqlite.JDBC	jdbc:sqlite:nombreBD

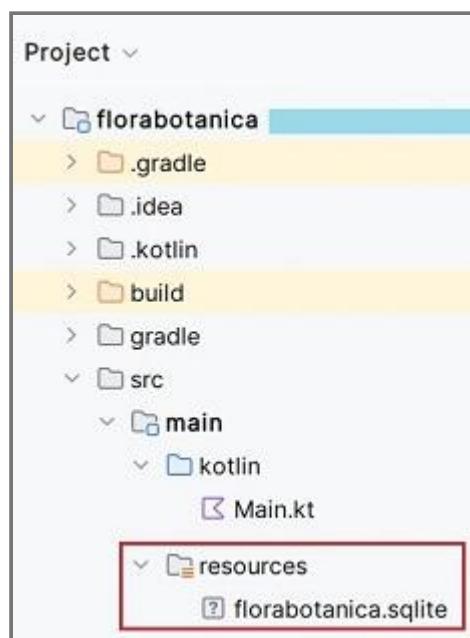
También dependiendo del SGBD será necesario utilizar la dependencia adecuada en **Gradle** añadiendo las líneas correspondientes en el fichero **build.gradle.kts**. A continuación se muestran las líneas para los SGBD PostgreSQL, MySQL y SQLite.

```
dependencies {
    implementation("org.postgresql:postgresql:42.7.1") //Postgres
    implementation("com.mysql:mysql-connector-j:8.3.0") //MySQL
    implementation("org.xerial:sqlite-jdbc:3.43.0.0") //SQLite
}
```

Hemos visto SQLite es una BD local y por tanto debe de estar dentro del proyecto, la ubicaremos en la carpeta `resources` ya que esta carpeta forma parte del classpath del proyecto y al compilarlo su contenido se copiará dentro del jar o build final. Así, si se comparte el proyecto, la BD viaja con él.

Ejemplo 1: Conexión a SQLite

El siguiente ejemplo muestra como conectar a una BD **SQLite** llamada `florabotanica.sqlite` que se encuentra en la carpeta `resources` dentro de un proyecto en **Kotlin**.



```

import java.io.File
import java.sql.DriverManager

fun main() {
    val dbPath = "src/main/resources/florabotanica.sqlite"
    println("Ruta de la BD: $dbPath")
    val url = "jdbc:sqlite:${dbPath}"
    DriverManager.getConnection(url).use { conn ->
        println("Conexión establecida correctamente")
    }
}

```



Prueba y analiza el ejemplo 1

1. Crea un proyecto kotlin con gradle o utiliza uno que ya tengas.
2. Añade las dependencias para trabajar con SQLite.
3. Descarga el fichero con la BD de ejemplo desde el siguiente enlace: [florabotanica.sqlite](#)
4. Copia el fichero en la carpeta correcta del proyecto.
5. Ejecuta el programa y comprueba que la salida por consola es la siguiente:

Ruta de la BD: F:\src\main\resources\florabotanica.sqlite

Conexión establecida correctamente con SQLite



Práctica 2: Crea tu proyecto Gradle y conecta a tu base de datos

1. Crea un nuevo proyecto en Kotlin con Gradle.
2. Añade las dependencias para trabajar con SQLite.
3. Copia la BD creada en la práctica anterior en la carpeta `resources`. Tu proyecto debe tener los mismos archivos que en la imagen del ejemplo anterior.
4. Añade las líneas de código necesarias para conectar con tu BD y muestra un mensaje indicando si se ha establecido la conexión correctamente o no.

2.1.3 2.3. Operaciones sobre la BD

En **JDBC** (Java Database Connectivity), las operaciones sobre la base de datos se realizan utilizando los siguientes objetos y métodos:

- **Connection**, establece el canal de comunicación con el SGBD (PostgreSQL, MySQL, etc.)
- Los objetos **PreparedStatement** y **CreateStatement** se utilizan para enviar consultas SQL desde el programa a la base de datos. A continuación se muestra una tabla con el uso de cada uno:

Si necesitas...	Usa...
Consultas sin parámetros	CreateStatement
Consultas con datos del usuario	PreparedStatement
Seguridad frente a inyecciones SQL	PreparedStatement
Ejecutar muchas veces con distintos valores	PreparedStatement
Crear tablas o sentencias SQL complejas que no cambian	CreateStatement

- Los métodos **executeQuery()**, **executeUpdate()** y **execute()** se utilizan para ejecutar sentencias SQL, pero se usan en contextos diferentes. A continuación se muestra una tabla con el uso de cada uno:

Método	Uso principal	Tipo de sentencia SQL	Resultado que devuelve
executeQuery()	Realizar consultas	SELECT	Objeto ResultSet con el resultado de la consulta SQL. Permite recorrer fila a fila el conjunto de resultados, accediendo a cada campo por nombre o por posición
executeUpdate()	Realizar modificaciones	INSERT, UPDATE, DELETE, DDL (CREATE, DROP, etc.)	Entero con el número de filas afectadas
execute()	No se sabe de antemano qué tipo de sentencia SQL se va a ejecutar (consulta o modificación)	Sentencias SQL que pueden devolver varios resultados	Booleano true si el resultado es un ResultSet (SELECT) y false si el resultado es un entero (INSERT, UPDATE, DELETE, CREATE, ALTER)

Liberación de recursos

Cuando una aplicación accede a una base de datos, abre varios recursos internos que consumen memoria y conexiones activas en el sistema:

- La conexión con el servidor de base de datos (Connection).
- Las sentencias SQL preparadas (Statement o PreparedStatement).
- El resultado de la consulta (ResultSet).

Estos recursos no se liberan automáticamente cuando se termina su uso (especialmente en Java o Kotlin con JDBC). Si no se cierran correctamente, se pueden producir problemas como:

- Fugas de memoria.
- Bloqueo de conexiones (demasiadas conexiones abiertas).
- Degradación del rendimiento.
- Errores inesperados en la aplicación.

Para liberar estos recursos hay dos opciones:

1. Usar try-catch-finally manual

Cuándo:

- No estás en Kotlin o no puedes usar .use.
- Necesitas capturar y manejar excepciones dentro del mismo método.
- Necesitas lógica extra antes o después de cerrar el recurso (por ejemplo, reintentos, logging detallado, liberar múltiples recursos en un orden específico).
- Estás trabajando en un proyecto que sigue un estilo más clásico de Java.

2. Utilización de .use { ... }

Es la que utilizaremos en nuestros proyectos.

Se recomienda utilizarlo si:

- Estás trabajando con un recurso que implementa AutoCloseable (Connection, Statement, ResultSet, File, etc.).
- Solo necesitas abrir, usar y cerrar el recurso de forma automática.
- No necesitas lógica compleja de manejo de excepciones dentro del mismo bloque.

Ventajas:

- Código más limpio y legible.
- Cierra automáticamente el recurso aunque ocurra una excepción.
- Evita errores de olvidar close().

Ejemplo 2: Utilización de close()

A continuación tienes un ejemplo en el que se declara una constante con la ruta a la BD, se establece la conexión, se consultan datos y se cierran los recursos abiertos (ResultSet, Statement y Connection) utilizando **close()** dentro de un bloque **finally** para garantizar su cierre incluso si ocurre un error. El orden correcto de cierre es del más interno al más externo:

```
import java.sql.Connection
import java.sql.Statement
import java.sql.ResultSet
import java.sql.DriverManager
import java.sql.SQLException

// Ruta al archivo de base de datos SQLite
const val URL_BD = "jdbc:sqlite:src/main/resources/florabotanica.sqlite"

fun main() {
    var conn: Connection? = null
    var stmt: Statement? = null
    var rs: ResultSet? = null

    try {
        conn = DriverManager.getConnection(URL_BD)
        println("Conectado a la BD")

        stmt = conn.createStatement()
        rs = stmt.executeQuery("SELECT * FROM plantas")

        while (rs.next()) {
            println(rs.getString("nombre_comun"))
        }
    } catch (e: SQLException) {
        println("Error al conectar o consultar la base de datos: ${e.message}")
    } catch (e: Exception) {
        e.printStackTrace()
    } finally {
        try {
            rs?.close()
            stmt?.close()
            conn?.close()
            println("Conexión cerrada correctamente")
        } catch (e: Exception) {
            println("Error al cerrar los recursos: ${e.message}")
        }
    }
}
```

```

    }
}
```

Prueba y analiza el ejemplo 2

Prueba el código de ejemplo y verifica que funciona correctamente.

Ejemplo 3: Utilización de .use

A continuación se muestra un **ejemplo con .use (sin necesidad de cerrar recursos manualmente)** que realiza la misma consulta que el ejemplo anterior. Ahora los recursos abiertos se cerrarán automáticamente. Además, por organización del código, se ha declarado una función para conectar a la BD:

- **conn.use { ... }** cierra la conexión automáticamente al final del bloque.
- **stmt.use { ... }** cierra el Statement automáticamente.
- **ResultSet** se cierra cuando cierras el Statement.

```

import java.sql.Connection
import java.sql.DriverManager
import java.sql.SQLException

// Ruta al archivo de base de datos SQLite
const val URL_BD = "jdbc:sqlite:src/main/resources/florabotanica.sqlite"

// Obtener conexión
fun conectarBD(): Connection? {
    return try {
        DriverManager.getConnection(URL_BD)
    } catch (e: SQLException) {
        e.printStackTrace()
        null
    }
}

fun main() {
    conectarBD()?.use { conn ->
        println("Conectado a la BD")

        conn.createStatement().use { stmt ->
            stmt.executeQuery("SELECT * FROM plantas").use { rs ->
                while (rs.next()) {
                    println(rs.getString("nombre_comun"))
                }
            }
        }
    } ?: println("No se pudo conectar")
}
```

Prueba y analiza el ejemplo 3

Prueba el código de ejemplo y verifica que funciona correctamente.

Práctica 3: Mejora tu proyecto

1. Declara una constante con la ruta a la BD.
2. Declara una función para conectar a la BD.
3. En el main conecta con la BD y realiza una consulta sobre tus datos utilizando .use (para no tener que cerrar recursos manualmente).

2.1.4 2.4. Objetos de acceso a datos (DAO)

Los objetos de acceso a datos son una buena forma de organizar nuestro código para manejar las diferentes operaciones CRUD de acceso a los datos. Es el Data Access Object (DAO) y algunas de las ventajas de utilizar estos objetos son las siguientes:

- Organización: todo el código SQL está en un único lugar.
- Reutilización: puedes llamar a PlantasDAO.listarPlantas() desde distintos sitios sin repetir la consulta.
- Mantenibilidad: si cambia la base de datos, solo tocas el DAO.
- Claridad: el resto de tu app se lee mucho más limpio, sin SQL mezclado.

Ejemplo 4: DAO

El siguiente ejemplo es el DAO para la tabla `plantas` de la BD `florabotanica.sqlite`. La estructura de la tabla `plantas` es la siguiente:

Name	Type	NN	PK	AI	U
<code>id_planta</code>	<code>INTEGER</code>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<code>nombre_comun</code>	<code>TEXT</code>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<code>nombre_cientifico</code>	<code>TEXT</code>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<code>stock</code>	<code>INTEGER</code>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<code>precio</code>	<code>REAL</code>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Creamos un archivo **PlantasDAO.kt** en el que declararemos una data class con la misma estructura que la tabla `plantas` y las funciones para leer la información de la tabla, añadir registros nuevos, modificar la información existente y borrarla. El código fuente es:

```

data class Planta(
    val id_planta: Int? = null, // lo genera SQLite automáticamente
    val nombreComun: String,
    val nombreCientifico: String,
    val stock: Int,
    val precio: Double
)

object PlantasDAO {
    fun listarPlantas(): List<Planta> {
        val lista = mutableListOf<Planta>()
        conectarBD()?.use { conn ->
            conn.createStatement().use { stmt ->
                stmt.executeQuery("SELECT * FROM plantas").use { rs ->
                    while (rs.next()) {
                        lista.add(
                            Planta(
                                id_planta = rs.getInt("id_planta"),
                                nombreComun = rs.getString("nombre_comun"),
                                nombreCientifico = rs.getString("nombre_cientifico"),
                                stock = rs.getInt("stock"),
                                precio = rs.getDouble("precio")
                            )
                        )
                    }
                }
            }
        } ?: println("No se pudo establecer la conexión.")
        return lista
    }

    // Consultar planta por ID
    fun consultarPlantaPorId(id: Int): Planta? {
        var planta: Planta? = null
        conectarBD()?.use { conn ->
            conn.prepareStatement("SELECT * FROM plantas WHERE id_planta = ?").use { pstmt ->
                pstmt.setInt(1, id)
                pstmt.executeQuery().use { rs ->
                    if (rs.next()) {
                        planta = Planta(
                            id_planta = rs.getInt("id_planta"),
                            nombreComun = rs.getString("nombre_comun"),
                            nombreCientifico = rs.getString("nombre_cientifico"),
                            stock = rs.getInt("stock"),
                            precio = rs.getDouble("precio")
                        )
                    }
                }
            }
        }
        return planta
    }
}

```

```

        )
    }
}
} ?: println("No se pudo establecer la conexión.")
return planta
}

fun insertarPlanta(planta: Planta) {
    conectarBD()?.use { conn ->
        conn.prepareStatement(
            "INSERT INTO plantas(nombre_comun, nombre_cientifico, stock, precio) VALUES (?, ?, ?, ?)"
        ).use { pstmt ->
            pstmt.setString(1, planta.nombreComun)
            pstmt.setString(2, planta.nombreCientifico)
            pstmt.setInt(3, planta.stock)
            pstmt.setDouble(4, planta.precio)
            pstmt.executeUpdate()
            println("Planta ${planta.nombreComun}' insertada con éxito.")
        } ?: println("No se pudo establecer la conexión.")
    }
}

fun actualizarPlanta(planta: Planta) {
    if (planta.id_planta == null) {
        println("No se puede actualizar una planta sin id.")
        return
    }
    conectarBD()?.use { conn ->
        conn.prepareStatement(
            "UPDATE plantas SET nombre_comun = ?, nombre_cientifico = ?, stock = ?, precio = ? WHERE id_planta = ?"
        ).use { pstmt ->
            pstmt.setString(1, planta.nombreComun)
            pstmt.setString(2, planta.nombreCientifico)
            pstmt.setInt(3, planta.stock)
            pstmt.setDouble(4, planta.precio)
            pstmt.setInt(5, planta.id_planta)
            val filas = pstmt.executeUpdate()
            if (filas > 0) {
                println("Planta con id=${planta.id_planta} actualizada con éxito.")
            } else {
                println("No se encontró ninguna planta con id=${planta.id_planta}.")
            }
        } ?: println("No se pudo establecer la conexión.")
    }
}

fun eliminarPlanta(id: Int) {
    conectarBD()?.use { conn ->
        conn.prepareStatement("DELETE FROM plantas WHERE id_planta = ?").use { pstmt ->
            pstmt.setInt(1, id)
            val filas = pstmt.executeUpdate()
            if (filas > 0) {
                println("Planta con id=$id eliminada correctamente.")
            } else {
                println("No se encontró ninguna planta con id=$id.")
            }
        } ?: println("No se pudo establecer la conexión.")
    }
}
}

```

La llamada a estas funciones desde **main.kt** podría ser:

```

fun main() {
    // Listar todas las plantas
    println("Lista de plantas:")
    PlantasDAO.listarPlantas().forEach {
        println("- [${it.id_planta}] ${it.nombreComun} (${it.nombreCientifico}), stock ${it.stock} unidades, precio: ${it.precio} €")
    }

    // Consultar planta por ID
    val planta = PlantasDAO.consultarPlantaPorId(3)
    if (planta != null) {
        println("Planta encontrada: [${planta.id_planta}] ${planta.nombreComun} (${planta.nombreCientifico}), stock ${planta.stock} unidades, precio: ${planta.precio} €")
    } else {
        println("No se encontró ninguna planta con ese ID.")
    }

    // Insertar plantas
    PlantasDAO.insertarPlanta(
        Planta(
            nombreComun = "Palmera",
            nombreCientifico = "Arecaceae",
            stock = 2,
            precio = 50.5
        )
    )

    // Actualizar planta con id=1
}

```

```

PlantasDAO.actualizarPlanta(
    Planta(
        id_planta = 1,
        nombreComun = "Aloe Arborescens",
        nombreCientifico = "Aloe barbadensis miller",
        stock = 20,
        precio = 5.8
    )
)

// Eliminar planta con id=2
PlantasDAO.eliminarPlanta(2)
}

```

✓ Prueba y analiza el ejemplo 4

Prueba el código de ejemplo y verifica que funciona correctamente.

⚠️ Práctica 4: Trabaja con tu base de datos

1. Añade a tu proyecto un objetos de acceso a datos (DAO) para manejar las diferentes operaciones CRUD de la primera tabla de tu BD.
2. Utiliza .use en todas tus operaciones para asegurarte de que se cierran correctamente todos los recursos.
3. Añade a tu proyecto un menú en tu función **main** para llamar a todas las operaciones CRUD que acabas de crear (pide la información por consola para las funciones que requieran el paso de información como parámetro) y comprueba que todas funcionan correctamente.
4. Añade otras dos tablas a tu BD y sus correspondientes DAO a tu proyecto.
5. Amplía el menú para poder gestionar los datos de todas las tablas.

2.1.5 2.5. Transacciones y excepciones

Transacciones

Una transacción es una secuencia de una o más operaciones sobre una base de datos que deben ejecutarse como una unidad indivisible. El objetivo es asegurar que todas las operaciones se completen con éxito o, en caso de fallo, ninguna de ellas se aplique, manteniendo así la base de datos en un estado consistente. Por ejemplo, en una transferencia bancaria, si falla el abono en una cuenta, se cancela el débito en la otra.

Las transacciones se gestionan mediante comandos como BEGIN TRANSACTION (para iniciar), COMMIT (para confirmar los cambios) y ROLLBACK (para deshacer los cambios en caso de error). Este mecanismo protege la base de datos frente a fallos parciales y situaciones de concurrencia, asegurando que los datos siempre reflejen una realidad válida y coherente.

Propiedades de una transacción (ACID)

Las transacciones garantizan propiedades fundamentales, conocidas por el acrónimo ACID:

Propiedad	Significado breve
Atomicidad	Todas las operaciones se ejecutan o ninguna lo hace
Consistencia	El sistema pasa de un estado válido a otro
Isolación	No interfiere con otras transacciones simultáneas
Durabilidad	Una vez confirmada, el cambio permanece

Comandos clave

Para controlar correctamente una transacción desde el código, necesitamos usar tres comandos clave:

- **commit()**: Confirma los cambios realizados por la transacción, haciéndolos permanentes.
- **rollback()**: Revierte todos los cambios realizados durante la transacción actual, volviendo al estado anterior.

Por defecto, muchas conexiones JDBC están en modo **auto-commit**, es decir, cada operación se ejecuta y confirma automáticamente. Para usar transacciones de forma manual, debes desactivar este modo:

```
conexion.setAutoCommit = false
```

Excepciones

El manejo de excepciones en las transacciones es absolutamente necesario para garantizar que los datos de la base de datos no queden en un estado inconsistente o corrupto cuando ocurre un error durante una operación.

Una transacción sin control de errores no es una transacción segura. Siempre hay que estar preparado para deshacer todo si algo sale mal.

Cuando realizamos varias operaciones dentro de una misma transacción (por ejemplo, una transferencia bancaria), pueden ocurrir errores como:

- un fallo de conexión,
- un ID incorrecto,
- un valor nulo inesperado,
- un error lógico como saldo insuficiente.

Si no controlamos esos errores, la base de datos podría:

- Aplicar solo algunas de las operaciones
- Dejar datos parcialmente modificados
- Generar resultados incorrectos para otros usuarios

Para evitarlo se utiliza un bloque **try-catch** que:

- Llama a commit() si todo sale bien
- Llama a rollback() si ocurre cualquier excepción

```
try {
    conexion.setAutoCommit = false

    // Varias operaciones SQL...
    conexion.commit() // Todo bien
} catch (e: Exception) {
    conexion.rollback() // Algo falló → revertir
    println("Error en la transacción. Cambios anulados.")
}
```

Ejemplo 5: commit y rollback

Para el siguiente ejemplo se han añadido a la BD las tablas `jardines` y `jardines_plantas` cuya estructura es la siguiente:

Name	Type	NN	PK	AI	U
id_jardin	INTEGER	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
nombre	TEXT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Name	Type	NN	PK	AI	U
id_jardin	INTEGER	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
id_planta	INTEGER	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
cantidad	INTEGER	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Supongamos que queremos llevar varias unidades de una planta a un jardín. El programa debe actualizar el stock en la tabla `plantas` (restando las unidades correspondientes) y añadir un registro en la tabla `jardines_plantas` indicando el jardín, la planta y la cantidad. Ambas operaciones deben realizarse juntas, o no realizarse ninguna. El código sería el siguiente:

```
fun llevarPlantasAJardin(id_jardin: Int, id_planta: Int, cantidad: Int) {
    conectarBD()?.use { conn ->
        try {
            conn.autoCommit = false // Iniciar transacción manual

            // Restar stock a la planta
            conn.prepareStatement("UPDATE plantas SET stock = stock - ? WHERE id_planta = ?").use { stock ->
                stock.setInt(1, id_planta)
                stock.executeUpdate()
            }

            // Añadir linea en tabla jardines_plantas
            conn.prepareStatement("INSERT INTO jardines_plantas(id_jardin, id_planta, cantidad) VALUES (?, ?, ?)").use { plantar ->
                plantar.setInt(1, id_jardin)
                plantar.setInt(2, id_planta)
                plantar.setInt(3, cantidad)
                plantar.executeUpdate()
            }

            // Confirmar cambios
            conn.commit()
            println("Transacción realizada con éxito.")
        } catch (e: SQLException) {
            if (e.message?.contains("UNIQUE constraint failed") == true) {
                println("Intento de insertar clave duplicada")
                conn.rollback()
                println("Transacción revertida.")
            } else {
                throw e // otros errores, relanzamos
            }
        } finally {
            // Código que se ejecuta siempre
            println("Fin del programa.")
        }
    }
}
```

Si no se produce ningún error se hará el `commit` y en caso contrario el `rollback`



Prueba y analiza el ejemplo 5

Prueba el código de ejemplo y verifica que funciona correctamente.



Práctica 5: Amplía tu proyecto

Incluye transacciones y control de errores mediante la captura de excepciones.



Entrega 1

Entrega en Aules la carpeta `main` de tu proyecto comprimida en formato `.zip`

IMPORTANTE: El proyecto no debe contener código que no se utilice, ni restos de pruebas de los ejemplos y no debe estar separado por prácticas. Debe ser un proyecto totalmente funcional.

2.1.6 2.6. Funciones y procedimientos almacenados

Las funciones (FUNCTION) y los procedimientos (PROCEDURE) **no se crean desde el lenguaje Kotlin**, ya que son elementos propios del SGBD. Para definirlos, se utiliza SQL y se ejecutan **directamente sobre la base de datos** a través de un cliente SQL.

Tanto las funciones como los procedimientos almacenados son bloques de código que se guardan en el servidor de la base de datos y que encapsulan una serie de instrucciones SQL.

Se usan para:

- Reutilizar operaciones complejas
- Organizar mejor la lógica de negocio
- Mejorar el rendimiento (menos tráfico entre app y BD)
- Mantener la integridad de datos

Concepto	Función	Procedimiento
Devuelve	Un valor simple	Un conjunto de datos o varios valores
Llamada SQL	SELECT fn_total_valor_planta(3)	CALL sp_listar_plantas_por_jardin(1)
Llamada en Kotlin	SELECT fn... CON PreparedStatement	CALL sp... CON CallableStatement
Uso típico	Cálculos	Listados, inserciones, actualizaciones

SQLite no soporta funciones ni procedimientos almacenados como lo hacen otros SGBD, por eso a partir de aquí seguiremos trabajando en MySQL.

⚠️ Práctica 6: Servidor MySQL

1. Monta tu servidor virtual siguiendo los pasos del documento [AWS Learner Lab](#) e instala en él tu servidor MySQL siguiendo los pasos del documento [Instalación MySQL en EC2](#)
2. Replica tu base de datos [SQLite](#) en [MySQL](#) dentro del servidor que acabas de crear. Puedes utilizar la herramienta [DBeaver](#) para crear las tablas e insertar los registros en ellas.
3. Haz una copia de tu proyecto y elimina todo lo relacionado con SQLite (incluido el archivo .sqlite).
4. Añade las líneas necesarias para conectar a tu BD MySQL.
5. Comprueba que la aplicación se está conectando a MySQL correctamente y que todas las opciones del menú siguen funcionando.

Funciones

Una **función** está diseñada para **calcular y devolver un resultado**. Se puede usar directamente dentro de una consulta SQL como parte de un SELECT, WHERE, ORDER BY, etc. Las funciones siempre devuelven un valor. La sintaxis general para crear una función en MySQL es la siguiente:

```
DELIMITER //
CREATE FUNCTION nombre_funcion(parámetro1 tipo, parámetro2 tipo, ...)
RETURNS tipo_dato
[DETERMINISTIC | NOT DETERMINISTIC]
[READS SQL DATA | MODIFIES SQL DATA | NO SQL]
BEGIN
    -- Declaraciones opcionales
    DECLARE variable_local tipo;

    -- Lógica de la función
    SET variable_local = ...;

    -- Retornar un valor
    RETURN variable_local;
END
//
```

```
DELIMITER ;
```

Parte	Significado
DELIMITER //	Cambia el delimitador temporalmente (porque dentro de la función usas ;).
CREATE FUNCTION nombre_funcion	Define la función y su nombre.
RETURNS tipo_dato	Especifica el tipo de valor que devolverá (INT , DOUBLE , VARCHAR(n) , etc.).
DETERMINISTIC	Indica que siempre devuelve el mismo resultado para los mismos parámetros. Esto permite que el optimizador de MySQL y los motores de replicación cacheen resultados o eviten reevaluaciones innecesarias.
NO DETERMINISTIC	Indica que el resultado puede variar aunque los argumentos sean iguales, por ejemplo si se usan funciones como RAND(), NOW(), etc.
BEGIN ... END	Marca el bloque de instrucciones.
DECLARE	Declara variables locales (opcional).
RETURN	Devuelve un único valor.
DELIMITER ;	Restablece el delimitador habitual.

Ejemplo 6: Trabajar con funciones

El siguiente ejemplo crea una función que devuelve el valor total del stock de una planta (stock × precio).

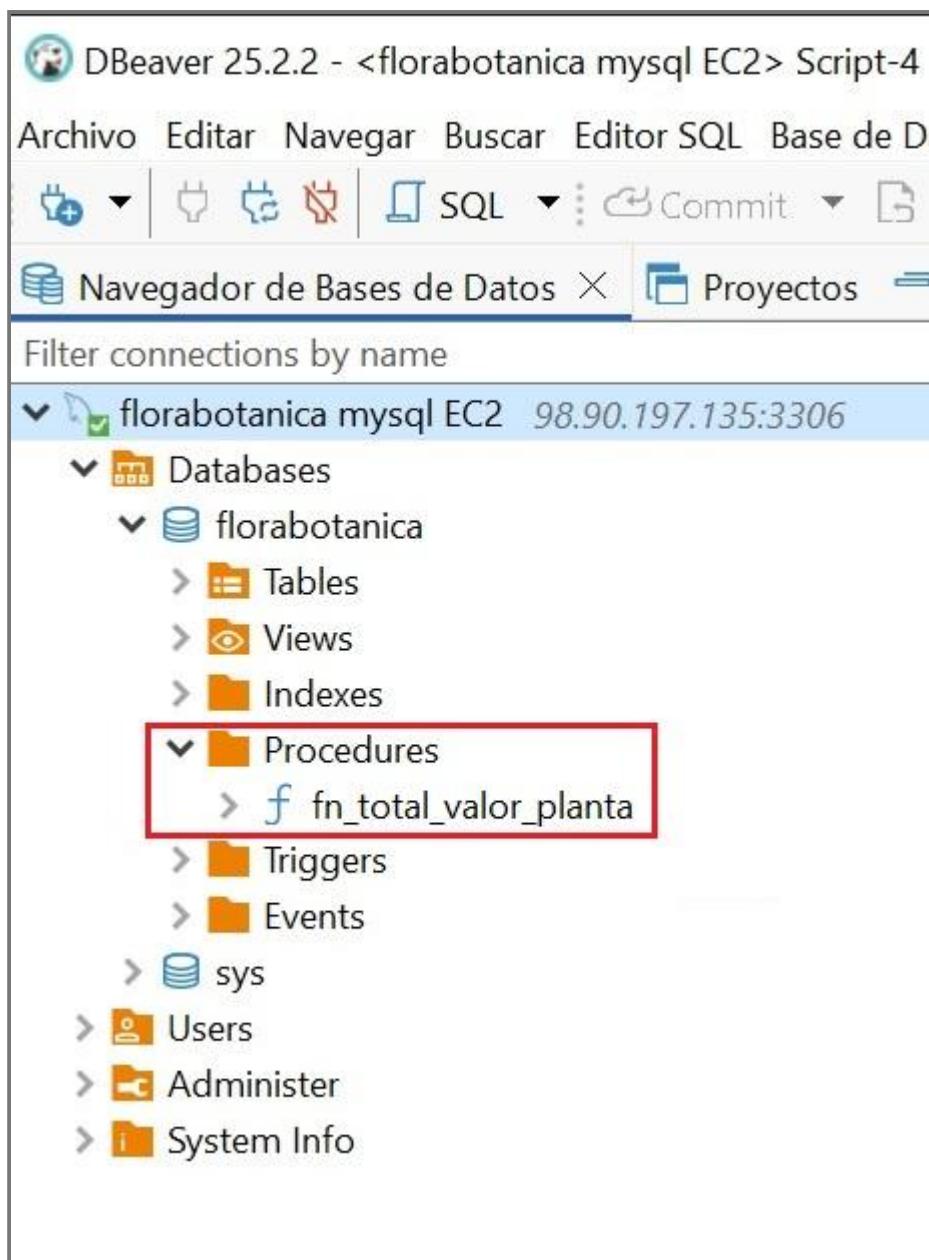
```
DELIMITER //
DROP FUNCTION IF EXISTS fn_total_valor_planta;
// 

CREATE FUNCTION fn_total_valor_planta(p_id_planta INT)
    RETURNS DOUBLE
    DETERMINISTIC
BEGIN
    DECLARE total DOUBLE;

    SET total = (
        SELECT stock * precio
        FROM plantas
        WHERE id_planta = p_id_planta);

    RETURN total;
END;
//
DELIMITER ;
```

Para que la función se guarde en la BD hay que ejecutar el código anterior como un script SQL. El resultado será el siguiente:



Una vez guardada, la podemos llamar desde dentro de la propia BD ejecutando el script SQL:

```
SELECT fn_total_valor_planta(3);
```

En este caso el resultado de la ejecución es el que se muestra en la siguiente imagen:

	123 fn_total_valor_planta(3)
1	137,6

Una vez que las funciones están creados en la base de datos, se pueden utilizar perfectamente desde Kotlin a través de JDBC, igual que se hace con cualquier consulta SQL y se gestionan mediante objetos `PreparedStatement`. Las funciones se invocan con `SELECT nombre_funcion(...)`. A continuación se muestra el código necesario para realizar la llamada desde Kotlin:

```
fun llamar_fn_total_valor_planta(id: Int){
    conectarBD()?.use { conn ->
        val sql = "SELECT fn_total_valor_planta(?)"
        conn.prepareStatement(sql).use { stmt ->
            stmt.setInt(1, id)
            stmt.executeQuery().use { rs ->
                if (rs.next()) {
                    val resultado = rs.getInt(1)
                    println("El valor es: $resultado")
                }
            }
        }
    }
}
```

✓ Prueba y analiza el ejemplo 6

Prueba el código de ejemplo y verifica que funciona correctamente.

⚠️ Práctica 7: Añade funciones a tu proyecto

1. Crea al menos dos funciones en tu base de datos y comprueba que se ejecutan correctamente desde dentro de ella.
2. Amplia el menú de tu proyecto y añade el código necesario para llamar a las funciones de tu BD.

Procedimientos

Un **procedimiento** sirve para **ejecutar acciones** dentro de la base de datos, como insertar registros, modificar datos o gestionar operaciones en bloque. La sintaxis general para crear un procedimiento en MySQL es la siguiente:

```
DELIMITER //
CREATE PROCEDURE nombre_procedimiento(
    [IN | OUT | INOUT] parametro1 tipo,
    [IN | OUT | INOUT] parametro2 tipo,
    ...
)
BEGIN
    -- Declaraciones opcionales
    DECLARE variable_local tipo;

    -- Lógica del procedimiento
    SELECT ...;
    UPDATE ...;
    -- etc.
END
//
```

```
DELIMITER ;
```

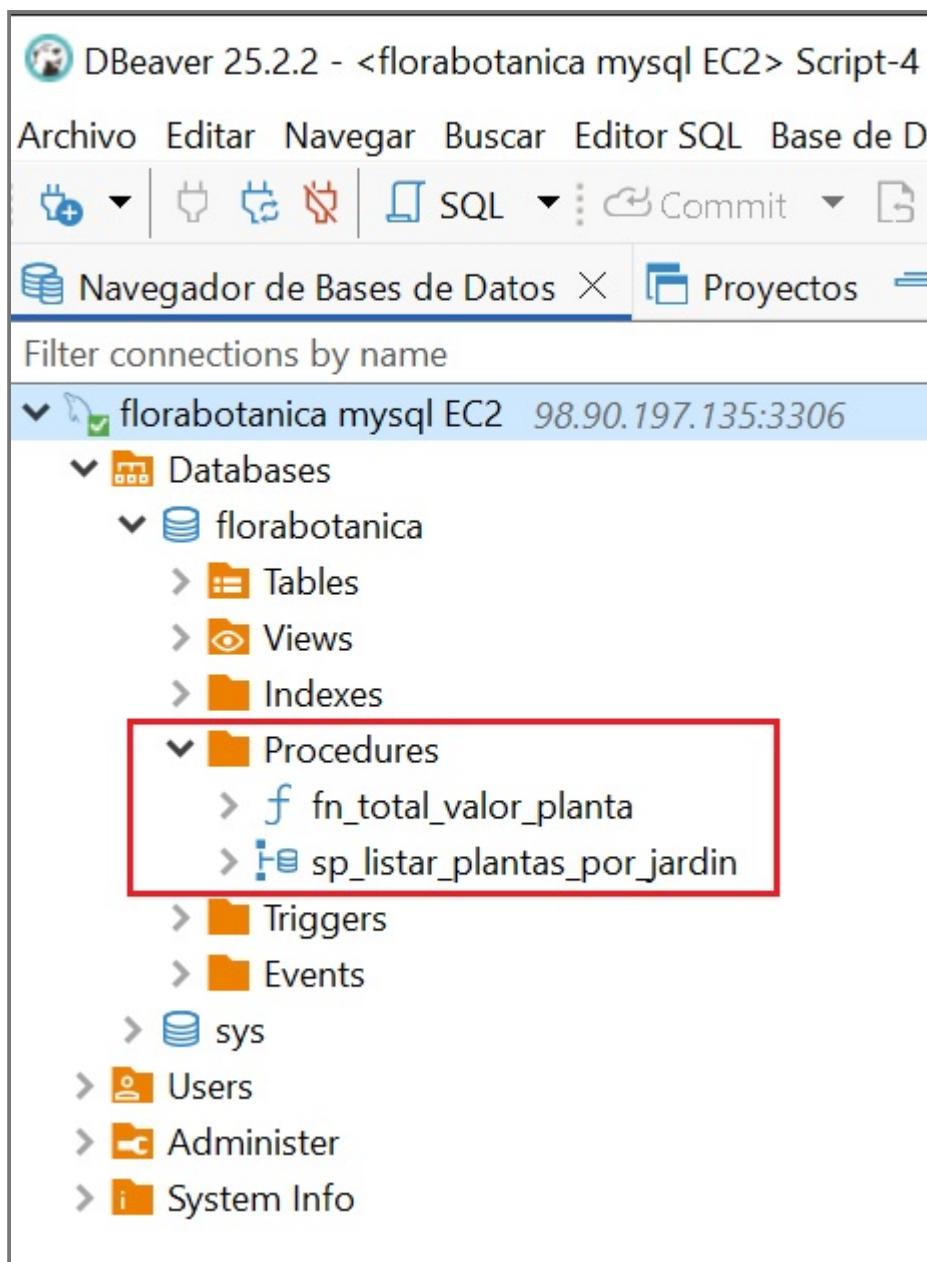
Parte	Descripción
DELIMITER //	Cambia el delimitador temporal para poder usar ; dentro del cuerpo.
CREATE PROCEDURE nombre	Declara el procedimiento.
IN , OUT , INOUT	Especifica la dirección del parámetro:
IN → se pasa al procedimiento (solo lectura).	
OUT → se devuelve como salida.	
INOUT → se pasa y puede ser modificado.	
BEGIN ... END	Define el bloque de instrucciones.
DECLARE	Declara variables locales si las necesitas.
DELIMITER ;	Restablece el delimitador normal.

Ejemplo 7: Trabajar con procedimientos

El siguiente ejemplo crea un procedimiento que devuelve un listado con las plantas y cantidades que hay en un jardín determinado.

```
DELIMITER //
DROP PROCEDURE IF EXISTS sp_listar_plantas_por_jardin;
// 
CREATE PROCEDURE sp_listar_plantas_por_jardin(IN p_id_jardin INT)
BEGIN
    SELECT j.nombre AS jardin,
           p.nombre_comun AS planta,
           jp.cantidad
      FROM jardines_plantas jp
     JOIN jardines j ON jp.id_jardin = j.id_jardin
     JOIN plantas p ON jp.id_planta = p.id_planta
    WHERE j.id_jardin = p_id_jardin;
END;
//
DELIMITER ;
```

Al igual que en las funciones, para que un procedimiento se guarde en la BD hay que ejecutar el código anterior como un script SQL. El resultado será el siguiente:



Una vez guardado, lo podemos llamar desde dentro de la propia BD ejecutando el script SQL siguiente:

```
CALL sp_listar_plantas_por_jardin(1);
```

En este caso el resultado de la ejecución es el que se muestra en la siguiente imagen:

The screenshot shows the MySQL Workbench interface. In the top query editor, a stored procedure is called: `CALL sp_listar_plantas_por_jardin(1);`. Below it, a table titled "jardines(+) 1" displays the results of a query: `SELECT fn_total_valor_planta(3)`. The table has three columns: "jardin" (id), "planta", and "cantidad". The data is as follows:

	AZ jardin	AZ planta	123 cantidad
1	Apolo	Aloe Arborescens	3
2	Apolo	Helecho de Boston	7
3	Apolo	Girasol	2
4	Apolo	Palmera	2
5	Apolo	prueba 10	2

Una vez que los procedimientos están creados en la base de datos, se pueden utilizar perfectamente desde Kotlin a través de JDBC, igual que se hace con cualquier consulta SQL y se gestionan mediante objetos `CallableStatement`. Los procedimientos se llaman con `CALL nombre_procedimiento(...)`. A continuación se muestra el código necesario para realizar la llamada desde Kotlin:

```
fun llamar_sp_listar_plantas_por_jardin(id: Int) {
    conectarBD()?.use { conn ->
        val sqlProcedimiento = "{CALL sp_listar_plantas_por_jardin(?)}"
        conn.prepareCall(sqlProcedimiento).use { call ->
            call.setInt(1, id) // id_jardin = 1
            call.executeQuery().use { rs ->
                println("\n Plantas del jardin :$id")
                while (rs.next()) {
                    val planta = rs.getString("planta")
                    val cantidad = rs.getInt("cantidad")
                    println(" - $planta (Cantidad: $cantidad)")
                }
            }
        }
    }
}
```

Prueba y analiza el ejemplo 7

Prueba el código de ejemplo y verifica que funciona correctamente.

Ejemplo 8: Otro ejemplo de procedimientos

El siguiente ejemplo crea un procedimiento que inserta una planta en un jardín (en la tabla `jardines_plantas`). El procedimiento recibe el `id_jardin`, el `id_planta` y una `cantidad`. Si la relación ya existe, actualizará la cantidad (sumando) y si no existe, insertará una nueva fila.

```
DELIMITER //
DROP PROCEDURE IF EXISTS sp_agregar_planta_a_jardin;
//CREATE PROCEDURE sp_agregar_planta_a_jardin(
-- IN p_id_jardin INT,
-- IN p_id_planta INT,
-- IN p_cantidad INT
--)
BEGIN
    -- Verificar si la relación jardin-planta ya existe
    IF EXISTS (
        SELECT 1 FROM jardines_plantas
        WHERE id_jardin = p_id_jardin AND id_planta = p_id_planta
    ) THEN
        -- Si existe, actualiza la cantidad
        UPDATE jardines_plantas
```

```

SET cantidad = cantidad + p_cantidad
WHERE id_jardin = p_id_jardin AND id_planta = p_id_planta;

SELECT CONCAT('Cantidad actualizada. Nueva cantidad: ', cantidad)
AS mensaje
FROM jardines_plantas
WHERE id_jardin = p_id_jardin AND id_planta = p_id_planta;

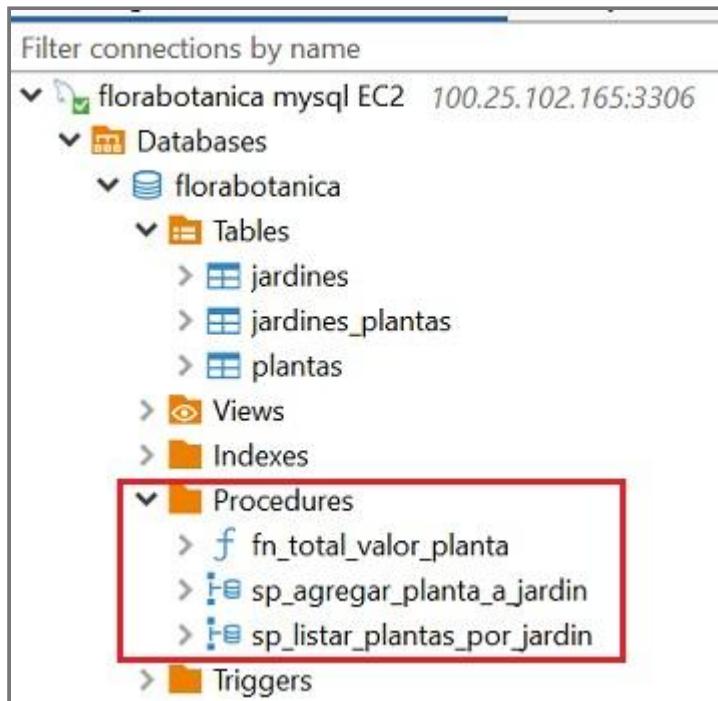
ELSE
-- Si no existe, inserta una nueva relación
INSERT INTO jardines_plantas (id_jardin, id_planta, cantidad)
VALUES (p_id_jardin, p_id_planta, p_cantidad);

SELECT 'Nueva planta agregada al jardín.' AS mensaje;
END IF;
END;
//

DELIMITER ;

```

Después de ejecutar el script anterior ya tenemos el procedimiento almacenado en nuestra BD:



Ejecutamos el script SQL dentro de la misma BD

```

-- Insertar una nueva planta en el jardín 2
CALL sp_agregar_planta_a_jardin(2, 5, 4);

-- Insertar más cantidad de una planta que ya existe
CALL sp_agregar_planta_a_jardin(2, 5, 3);

-- Verificar resultado
SELECT * FROM jardines_plantas WHERE id_jardin = 2 AND id_planta = 5;

```

El resultado de la ejecución es el que se muestra en la siguiente imagen:

```

florabotanica mysql EC2 > Script-8 <
-- Insertar una nueva planta en el jardín 2
CALL sp_agregar_planta_a_jardin(2, 5, 4);

-- Insertar más cantidad de una planta que ya existe
CALL sp_agregar_planta_a_jardin(2, 5, 3);

-- Verificar resultado
SELECT * FROM jardines_plantas WHERE id_jardin = 2 AND id_planta = 5;

```

	id_jardin	id_planta	cantidad
1	2	5	7

A continuación se muestra el código necesario para realizar la llamada desde Kotlin:

```

fun llamar_sp_agregar_planta_a_jardin(id_p:Int, id_j:Int, cant:Int){
    conectarBD()?.use { conn ->
        val sql = "{CALL sp_agregar_planta_a_jardin(?, ?, ?)}"
        conn.prepareStatement(sql).use { call ->
            call.setInt(1, id_p) // id_jardin
            call.setInt(2, id_j) // id_planta
            call.setInt(3, cant) // cantidad

            call.executeQuery().use { rs ->
                while (rs.next()) {
                    println(rs.getString("mensaje"))
                }
            }
        }
    }
}

```

✓ Prueba y analiza el ejemplo 8

Prueba el código de ejemplo y verifica que funciona correctamente.

⚠️ Práctica 8: Añade procedimientos a tu proyecto

1. Crea al menos dos procedimientos, uno que devuelva información resultante de realizar una consulta entre todas las tablas que hay en tu BD y otro que inserte información de una de las tablas.
2. Amplia el menú de tu proyecto y añade el código necesario para llamar a los procedimientos de tu BD.

⚡ Entrega 2

Realiza lo siguiente:

1. Exporta tu BD con el comando `mysqldump` en formato `.sql` (Puedes consultar el documento [Instalación de MySQL en EC2](#) apartado [Exportación de la BD](#)).
2. Copia el archivo `.sql` a la carpeta `resources` de tu proyecto.

Entrega en Aules la carpeta `main` de tu proyecto comprimida en formato `.zip`

IMPORTANTE: El proyecto no debe contener código que no se utilice, ni restos de pruebas de los ejemplos y no debe estar separado por prácticas. Debe ser un proyecto totalmente funcional.

Autoría

Obra realizada por Begoña Paterna Lluch basada en materiales desarrollados por Alicia Salvador Contreras. Publicada bajo licencia [Creative Commons Atribución/Reconocimiento-CompartirIgual 4.0 Internacional](#)

2.2 Unidad 3. Acceso a Bases de Datos documentales

Revisiones

Revisión	Fecha	Descripción
1.0	31-10-2025	Adaptación de los materiales a markdown
1.1	28-11-2025	Modificación de funciones de importación y exportación

2.2.1 3.1. Introducción

Las bases de datos documentales nativas (como MongoDB, Redis o Firebase) almacenan información en forma de documentos, usualmente codificados en JSON, BSON o XML, en lugar de filas y columnas como en las bases de datos relacionales.

Cada documento puede tener una estructura diferente, lo que permite mayor flexibilidad y agilidad en el desarrollo.

Sin embargo, si el dominio de la aplicación tiene muchas relaciones fuertes entre entidades y se necesita garantizar una integridad referencial estricta, una base de datos relacional puede ser más adecuada.

Ventajas

Ventaja	Descripción
Flexibilidad del esquema	No es necesario definir un esquema fijo antes de insertar datos. Ideal para estructuras dinámicas.
Escalabilidad horizontal	Se adaptan bien al escalado distribuyendo los datos en múltiples servidores (sharding).
Rendimiento en lectura y escritura	Muy eficiente en operaciones de lectura y escritura sobre documentos completos.
Modelo cercano a objetos	Almacenan los datos de manera similar a como se manejan en el código (objetos serializados como JSON).
Facilidad de integración con APIs REST	Los documentos JSON pueden ser enviados y recibidos fácilmente a través de APIs REST.
Ideal para datos semiestructurados	Útiles para trabajar con datos que no se ajustan a una estructura tabular, como respuestas de formularios, logs, etc.

Inconvenientes

Inconveniente	Descripción
Falta de integridad referencial	No hay claves foráneas como en las bases de datos relacionales, lo que puede causar inconsistencias si no se gestiona adecuadamente desde la aplicación.
Redundancia de datos	Se repite información entre documentos al no haber normalización; esto puede generar más uso de espacio.
Curva de aprendizaje	Requiere aprender nuevos conceptos como agregaciones, operadores específicos y estructuras de documentos.
Menor soporte para transacciones complejas	Aunque existen transacciones en algunas bases (como MongoDB), su uso es más limitado que en sistemas relacionales.
Consultas menos optimizadas en relaciones complejas	No es la mejor opción cuando los datos necesitan muchas relaciones y joins complejos.

Estructuras básicas de almacenamiento de información

Concepto	Equivalente en BD relacional	Descripción
Base de datos	Base de datos	Conjunto de colecciones.
Colección	Tabla	Agrupación de documentos relacionados.
Documento	Fila (registro)	Unidad básica de almacenamiento. Es un objeto JSON.
Campo	Columna	Atributo dentro del documento.

2.2.2 3.2. JSON

JSON (JavaScript Object Notation) es un formato de texto ligero utilizado para almacenar e intercambiar información estructurada entre aplicaciones. Aunque su sintaxis proviene de JavaScript, hoy en día es independiente del lenguaje y se usa ampliamente en entornos como Kotlin, Java, Python, Node.js, bases de datos NoSQL, APIs REST, etc.

Un fichero JSON está compuesto por **pares clave-valor**, donde:

- Las **claves** siempre van entre comillas dobles " ".
- Los **valores** pueden ser:
 - Cadenas de texto ("texto")
 - Números (42)
 - Booleanos (true o false)
 - Objetos (otro conjunto de pares clave-valor { ... })
 - Arrays o listas ([...])
 - Valor nulo (null)

Ejemplos de estructuras JSON

Objeto simple: Representa un único elemento con propiedades básicas.

```
{
  "nombre": "Pol",
  "edad": 21,
  "ciudad": "Castellón"
}
```

Objeto con array(lista de valores): Incluye un campo que contiene una lista.

```
{
  "nombre": "Pol",
  "aficiones": ["libros", "cine", "música"]
}
```

Objeto con otro objeto anidado: Un campo puede contener a su vez otro objeto JSON.

```
{
  "nombre": "Pol",
  "edad": 21,
  "dirección": {
    "calle": "Mayor",
    "ciudad": "Castellón",
    "codigo_postal": 12001
  }
}
```

Array de objetos: Cuando necesitamos almacenar varios elementos similares (por ejemplo, una lista de productos o alumnos).

```
{
  "alumnos": [
    { "nombre": "Pol", "nota": 7.6 },
    { "nombre": "Eli", "nota": 8.2 },
    { "nombre": "Mar", "nota": 9.8 }
  ]
}
```

Combinación compleja (objetos + arrays + anidamientos): Para representar datos estructurados, como los de una tienda online.

```
{
  "pedido": {
    "id": 101,
    "fecha": "2025-10-11",
    "cliente": {
      "nombre": "Pol",
      "email": "pol@dominio.com"
    },
    "productos": [
      { "nombre": "Ensalada de piña", "precio": 10.50, "cantidad": 1 },
      { "nombre": "Tarta de manzana", "precio": 3.50, "cantidad": 1 }
    ],
    "total": 14.00
  }
}
```

Array de objetos principales: También se puede usar un array como estructura raíz, por ejemplo, para representar varios registros en un mismo fichero:

```
[
  { "nombre": "Pol", "edad": 21 },
  { "nombre": "Eli", "edad": 22 },
  { "nombre": "Mar", "edad": 18 }
]
```

⚠️ Práctica 1: Crea y valida tu JSON

1. Crea un fichero `datos.json` con la información con la que estás trabajando.
2. Asegúrate de incluir diferentes tipos de datos (texto, número, booleano, array y objeto anidado).
3. Valida el archivo utilizando <https://jsonlint.com>

2.2.3 3.3. MongoDB

Introducción

MongoDB es un sistema de gestión de bases de datos NoSQL **orientado a documentos**. A diferencia de las bases de datos relacionales, que almacenan la información en tablas con filas y columnas, MongoDB guarda los datos en **colecciones** formadas por documentos en formato **BSON** (una representación binaria de JSON).

En **MongoDB** cada documento es una **estructura flexible**, parecida a un objeto de programación, donde los datos se organizan en pares **clave-valor**. Esta flexibilidad permite que cada **documento** tenga una estructura diferente, lo que hace que MongoDB se adapte fácilmente a los cambios en los datos sin necesidad de modificar esquemas.

Ejemplo 1: Plantas y jardineros

A continuación tenemos información sobre algunas **plantas** y los **jardineros** que las cuidan. Dependiendo de cómo se deba acceder a la información, se pueden guardar las plantas con sus jardineros, o los jardineros con las plantas que cuidan.

De la primera manera podríamos tener una colección llamada **Plantas**. Observa cómo los objetos no tienen por qué tener la misma estructura y, en este caso, la forma de acceder al nombre de un jardinero sería la siguiente: **objeto.jardinero.nombre**:

```
{
  "id_planta": 301,
  "nombre": "Rosa silvestre",
  "tipo": "Arbusto",
  "jardinero": {
    "nombre": "Eli",
    "apellidos": "Martínez Serra",
    "anyo_nacimiento": 1985
  },
  "localitzacio": "Jardí Mediterrani"
},
{
  "id_planta": 302,
  "nombre": "Ficus lyrata",
  "tipo": "Planta de interior",
  "jardinero": {
    "nombre": "Pol",
    "apellidos": "García Pérez",
    "anyo_nacimiento": 1990
  }
}
```

```

        "apellidos": "Ribas Colomer",
        "pais": "Espanya"
    },
    "altura": 150,
    "riego_semanal": 2
}

```

De la segunda manera tendríamos la colección **Jardineros** donde la información estaría organizada por jardineros y cada uno de ellos tendría un array con las plantas que cuida (los corchetes: []):

```

{
    "id_jardinero": 401,
    "nombre": "Eli",
    "apellidos": "Martinez Serra",
    "anyo_nacimiento": 1985,
    "plantas": [
        {
            "nombre": "Rosa silvestre",
            "tipo": "Arbust",
            "ubicacion": "Jardi Mediterrani"
        },
        {
            "nombre": "Lavanda officinalis",
            "ubicacion": "Parterre aromàtic"
        }
    ],
    {
        "id_jardinero": 402,
        "nombre": "Pol",
        "apellidos": "Ribas Colomer",
        "pais": "España",
        "plantas": [
            {
                "nombre": "Ficus lyrata",
                "tipo": "Planta d'interior",
                "altura": 150,
                "riego_semanal": 2
            }
        ]
    }
}

```

Práctica 2: Amplía tu JSON

1. Amplía el JSON de la práctica anterior para que contenga información estructurada como la del ejemplo anterior (utilizando uno de los dos ejemplos).
2. Valida el archivo utilizando <https://jsonlint.com>

Trabajar con MongoDB

MongoDB utiliza su propia **shell interactiva**, llamada `mongosh`, que permite ejecutar comandos para administrar bases de datos, colecciones y documentos. Su sintaxis es **muy similar a JavaScript**, ya que cada comando se ejecuta sobre un **objeto base**:

```
db.coleccion.operacion()
```

- `db` → representa la base de datos actual.
- `colección` → el nombre de la colección sobre la que actuamos.
- `operacion()` → el comando que deseamos ejecutar.

En cualquier operación, debemos escribir `db` seguido del nombre de la colección y después la operación a realizar. Para guardar un documento ejecutamos el siguiente comando:

```
db.ejemplo.insertOne({ msg: "Hola, ¿qué tal?" })
```

Obtendremos una respuesta indicando que se ha insertado un documento en la **colección ejemplo** (si no existía, la crearía automáticamente):

```
{
    acknowledged: true,
    insertedId: ObjectId('68ff6004ab24a06f35cebea4')
}
```

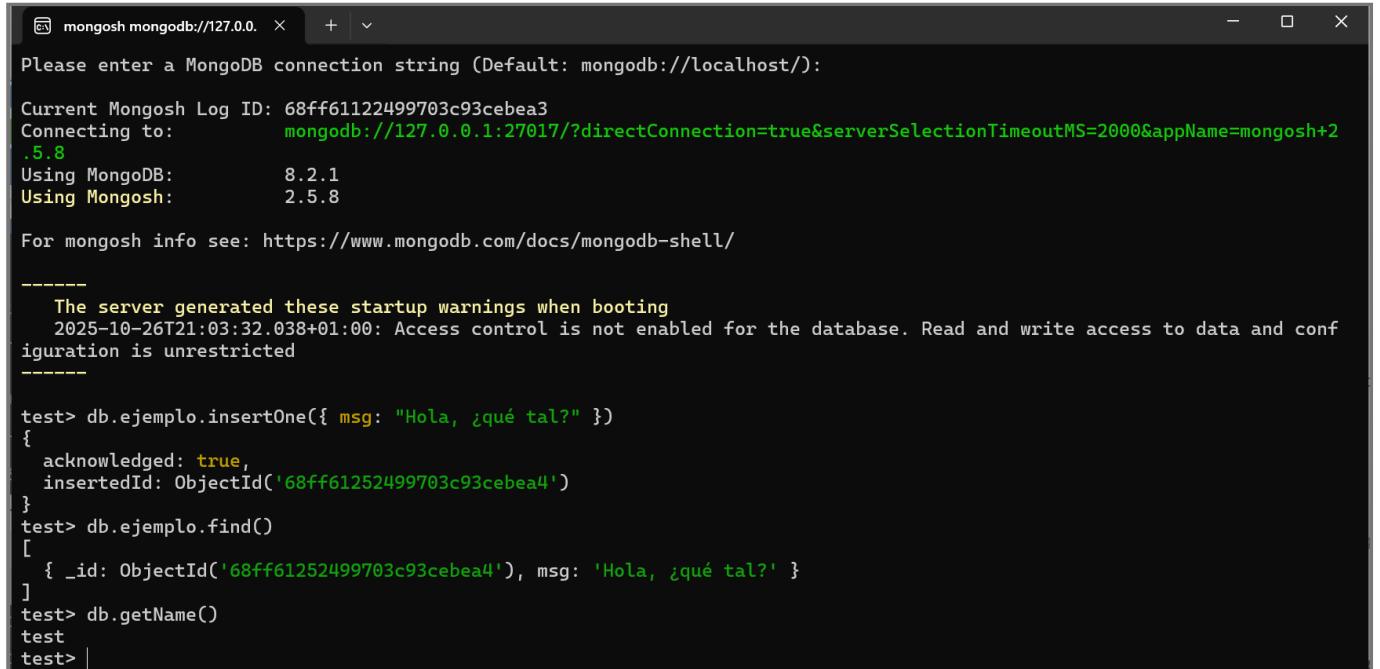
Y con el siguiente comando recuperamos la información:

```
db.ejemplo.find()
```

Lo que nos devolverá algo como:

```
{ "_id" : ObjectId("56cc1acd73b559230de8f71b"), "msg" : "Hola, ¿qué tal?" }
```

Todo esto se realiza en la misma terminal, y cada uno de nosotros obtendrá un número diferente en el campo **ObjectId**. En la siguiente imagen pueden verse las dos operaciones.



```
mongosh mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.5.8
Please enter a MongoDB connection string (Default: mongodb://localhost/):
Current Mongosh Log ID: 68ff61122499703c93cebea3
Connecting to: mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.5.8
Using MongoDB: 8.2.1
Using Mongosh: 2.5.8

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/
-----
The server generated these startup warnings when booting
2025-10-26T21:03:32.038+01:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
-----

test> db.ejemplo.insertOne({ msg: "Hola, ¿qué tal?" })
{
  acknowledged: true,
  insertedId: ObjectId('68ff61252499703c93cebea4')
}
test> db.ejemplo.find()
[
  { _id: ObjectId('68ff61252499703c93cebea4'), msg: 'Hola, ¿qué tal?' }
]
test> db.getName()
test
test> |
```

Información útil del entorno

Comando	Descripción
db.stats()	Muestra estadísticas sobre la base de datos. Ejemplo: db.stats()
db.coleccion.stats()	Muestra estadísticas sobre una colección. Ejemplo: db.alumnos.stats()
db.version()	Devuelve la versión de MongoDB. Ejemplo: db.version()

Comandos sobre bases de datos

Comando	Descripción	Ejemplo
show dbs	Muestra todas las bases de datos existentes.	show dbs
use <nombre>	Cambia a una base de datos (la crea si no existe).	use biblioteca
db.getName()	Muestra el nombre de la base de datos actual.	db.getName()
db.dropDatabase()	Elimina la base de datos actual.	db.dropDatabase()

Comandos sobre colecciones

Comando	Descripción
<code>show collections</code>	Lista todas las colecciones de la base de datos. Ejemplo: <code>show collections</code>
<code>db.createCollection("nombre")</code>	Crea una colección vacía. Ejemplo: <code>db.createCollection("alumnos")</code>
<code>db.coleccion.drop()</code>	Elimina una colección completa. Ejemplo: <code>db.alumnos.drop()</code>
<code>db.coleccion.renameCollection("nuevoNombre")</code>	Cambia el nombre de una colección. Ejemplo: <code>db.alumnos.renameCollection("estudiantes")</code>

⚠️ Práctica 3: Instala MongoDB

Instala MongoDB en tu ordenador siguiendo la guía [Instalación y administración de MongoDB](#).

Ejemplo 2: Crear BD, insertar plantas y mostrarlas

El siguiente ejemplo crea una base de datos llamada `florabotanica`. Crea una colección llamada `plantas` e inserta tres documentos con campos: `nombre_comun`, `nombre_cientifico`, `altura`. Por último muestra todas las bases de datos y las colecciones creadas.

```
// Abrir mongosh
mongosh

// Crear/usar la base de datos
use florabotanica

// Insertar documentos (si la colección no existe, se crea automáticamente)
db.plantas.insertMany([
  { id_planta: 1, nombre_comun: "Aloe", nombre_cientifico: "Aloe vera", altura: 30 },
  { id_planta: 2, nombre_comun: "Pino", nombre_cientifico: "Pinus sylvestris", altura: 330 },
  { id_planta: 3, nombre_comun: "Cactus", nombre_cientifico: "Cactaceae", altura: 120 }
])

// Comprobar bases de datos y colecciones
show dbs
show collections
```

El ejemplo funciona de la siguiente manera:

- `use florabotanica` cambia el contexto
- `insertMany` inserta varios documentos
- `show dbs` no mostrará `florabotanica` hasta que la colección tenga datos persistidos;
- tras insertar, aparecerá en la lista

Salida esperada:

```
{ acknowledged: true, insertedIds: { '0': ObjectId(...), '1': ObjectId(...), '2': ObjectId(...) } }

> show dbs
admin 0.000GB
config 0.000GB
local 0.000GB
florabotanica 0.001GB

> show collections
plantas
```

✓ Prueba y analiza el ejemplo 2

Prueba el código de ejemplo y verifica que funciona correctamente.

⚠️ Práctica 4: Crea tu BD, inserta y muestra información

1. Abre la terminal (`mongosh`) y crea tu BD.
2. Crea una colección e inserta tres documentos con los campos que quieras.
3. Muestra todas las bases de datos y las colecciones creadas.

Una vez comprendido el manejo desde terminal, trabajaremos con kotlin a través del *driver oficial de MongoDB para Kotlin*. Para ello crearemos un nuevo proyecto en IntelliJ con Gradle. Además, para los ejemplos realizados en Kotlin de esta unidad, se han declarado tres constantes para almacenar el servidor, el nombre de la BD y el nombre de la colección con los que vamos a trabajar. También se crea una variable Scanner de forma global para poder utilizarla en cualquier parte del programa.

```
const val NOM_SRV = "mongodb://localhost:27017"
const val NOM_BD = "florabotanica"
const val NOM_COLECCION = "plantas"

// Creamos el Scanner de forma global
val scanner = Scanner(System.`in`)
```

Ejemplo 3: Conexión y lectura de información en Kotlin

El siguiente ejemplo añade la dependencia del driver de MongoDB, conecta a la BD `florabotanica` y muestra por consola la información de cada documento JSON almacenado en `plantas`.

1. Añadir dependencia al fichero `build.gradle.kts`

```
implementation("org.mongodb:mongodb-driver-sync:4.11.0")
```

2. Conectar a la BD y leer la información

```
import com.mongodb.client.MongoClients
import com.mongodb.client.model.Filters
import com.mongodb.client.model.Updates
import com.mongodb.client.model.UpdateOptions
import com.mongodb.client.model.Sorts

fun mostrarPlantas() {
    val cliente = MongoClients.create(NOM_SRV)
    val db = cliente.getDatabase(NOM_BD)
    val colección = db.getCollection(NOM_COLECCION)

    // Mostrar documentos de la colección plantas
    val cursor = colección.find().iterator()
    cursor.use {
        while (it.hasNext()) {
            val doc = it.next()
            println(doc.toJson())
        }
    }
    cliente.close()
}
```

✓ Prueba y analiza el ejemplo 3

Prueba el código de ejemplo y verifica que funciona correctamente.

⚠️ Práctica 5: Trabaja con tu BD

1. Crea un proyecto Kotlin en IntelliJ IDEA.
2. Añade la dependencia del driver de MongoDB (`org.mongodb:mongodb-driver-sync`).
3. Conéctate a tu base de datos y muestra los documentos de tu colección. Intenta formatear la información que sale por consola. Por ejemplo:

```
**** Listado de plantas:
[1] Aloe (Aloe vera): 30 cm
[2] Pino (Pinus sylvestris): 330 cm
[3] Cactus (Cactaceae): 120 cm
```

Operaciones básicas

Inserción (Si la colección no existe, MongoDB la **creará automáticamente** en el momento de la inserción)

Comando	Descripción
<code>insertOne()</code>	Inserta un solo documento. Ejemplo: <code>db.alumnos.insertOne({nombre:"Ana", nota:8})</code>
<code>insertMany()</code>	Inserta varios documentos a la vez. Ejemplo: <code>db.alumnos.insertMany([{nombre:"Luis", nota:7}, {nombre:"Marta", nota:9}])</code>

Búsqueda

Comando	Descripción
<code>find()</code>	Devuelve todos los documentos de la colección. Ejemplo: <code>db.alumnos.find()</code>
<code>findOne()</code>	Devuelve el primer documento que cumple una condición. Ejemplo: <code>db.alumnos.findOne({nombre:"Ana"})</code>
<code>find(criterio, proyección)</code>	Permite filtrar y mostrar solo algunos campos. Ejemplo: <code>db.alumnos.find({nota:{\$gte:8}}, {nombre:1, _id:0})</code>

Operadores comunes:

`$eq` (igual), `$ne` (distinto), `$gt` (mayor que), `$lt` (menor que), `$gte` (mayor o igual), `$lte` (menor o igual), `$in`, `$and`, `$or`.

Actualización (Usa `$set` para modificar solo algunos campos y **no perder el resto**)

Comando	Descripción
<code>updateOne(filtro, cambios)</code>	Actualiza el primer documento que cumpla la condición. Ejemplo: <code>db.alumnos.updateOne({nombre:"Ana"}, {\$set:{nota:9}})</code>
<code>updateMany(filtro, cambios)</code>	Actualiza todos los documentos que cumplan la condición. Ejemplo: <code>db.alumnos.updateMany({nota:{\$lt:5}}, {\$set:{aprobado:false}})</code>
<code>replaceOne(filtro, nuevoDoc)</code>	Sustituye el documento completo. Ejemplo: <code>db.alumnos.replaceOne({nombre:"Ana"}, {nombre:"Ana", nota:10})</code>

Eliminación

Comando	Descripción
<code>deleteOne()</code>	Elimina el primer documento que cumpla la condición. Ejemplo: <code>db.alumnos.deleteOne({nombre:"Luis"})</code>
<code>deleteMany()</code>	Elimina todos los documentos que cumplan la condición. Ejemplo: <code>db.alumnos.deleteMany({nota:{\$lt:5}})</code>

Ejemplo 4: Operaciones CRUD en terminal

El siguiente ejemplo realiza las siguientes operaciones sobre la colección `plantas`:

1. Inserta tres nuevos documentos con `insertMany()`.
2. Recupera todos los documentos con `find()`.
3. Filtra aquellos cuya `altura` sea mayor de 100.
4. Actualiza uno de los documentos cambiando la altura.
5. Elimina una planta específica mediante `deleteOne()`.

```
// 1) Insertar tres nuevos documentos
db.plantas.insertMany([
  { id_planta: 4, nombre_comun: "Lavanda", nombre_cientifico: "Lavandula", altura: 50, tipo: "arbusto" },
  { id_planta: 5, nombre_comun: "Rosal", nombre_cientifico: "Rosa", altura: 120, tipo: "arbusto" },
  { id_planta: 6, nombre_comun: "Olivo", nombre_cientifico: "Olea europaea", altura: 800, tipo: "árbol" }
])

// 2) Recuperar todos los documentos
db.plantas.find().pretty()

// 3) Filtrar altura > 100
db.plantas.find({ altura: { $gt: 100 } }).pretty()

// 4) Actualizar: cambiar altura de "Cactus" a 130
db.plantas.updateOne({ nombre_comun: "Cactus" }, { $set: { altura: 130 } })

// 5) Eliminar una planta por nombre
db.plantas.deleteOne({ nombre_comun: "Rosal" })
```

El ejemplo funciona de la siguiente manera:

- `insertMany` devuelve `acknowledged: true` con `insertedIds`.
- `find().pretty()` muestra documentos en JSON formateado.
- `find({ altura: { $gt: 100 } })` listará pinos, olivos, etc.
- `updateOne` devuelve un objeto con `matchedCount` y `modifiedCount`.
- `deleteOne` devuelve `deletedCount: 1` si eliminó un documento.

Salida de `updateOne`:

```
{ acknowledged: true, matchedCount: 1, modifiedCount: 1, upsertedId: null }
```

Salida de `deleteOne`:

```
{ acknowledged: true, deletedCount: 1 }
```



Prueba y analiza el ejemplo 4

Prueba el código de ejemplo y verifica que funciona correctamente.



Práctica 6: Trabaja con tu BD

1. Inserta tres nuevos documentos con `insertMany()`.
2. Recupera todos los documentos con `find()`.
3. Aplica algún filtro.
4. Actualiza uno de los documentos cambiando el valor de un campo.
5. Elimina un documento específico mediante `deleteOne()`.

Ejemplo 5: Operaciones CRUD desde Kotlin

El siguiente fragmento de código realiza las siguientes operaciones sobre la colección `plantas` de la BD `florabotanica`:

1. Insertar un nuevo documento a partir de los datos introducidos por el usuario.
2. Actualizar la altura de una planta dada.
3. Eliminar una planta por nombre.

```

import com.mongodb.client.MongoClients
import com.mongodb.client.MongoDatabase
import com.mongodb.client.model.Filters
import org.bson.Document
import java.util.Scanner

fun insertarPlanta() {
    //conectar con la BD
    val cliente = MongoClients.create(NOM_SRV)
    val db = cliente.getDatabase(NOM_BD)
    val colección = db.getCollection(NOM_COLECCION)

    var id_planta: Int? = null
    while (id_planta == null) {
        print("ID de la planta: ")
        val entrada = scanner.nextLine()
        id_planta = entrada.toIntOrNull()
        if (id_planta == null) {
            println("El ID debe ser un número !!!")
        }
    }

    print("Nombre común: ")
    val nombre_común = scanner.nextLine()
    print("Nombre científico: ")
    val nombre_científico = scanner.nextLine()

    var altura: Int? = null
    while (altura == null) {
        print("Altura (en cm): ")
        val entrada = scanner.nextLine()
        altura = entrada.toIntOrNull()
        if (altura == null) {
            println("La altura debe ser un número !!!")
        }
    }

    val doc = Document("id_planta", id_planta)
        .append("nombre_común", nombre_común)
        .append("nombre_científico", nombre_científico)
        .append("altura", altura)

    colección.insertOne(doc)
    println("Planta insertada con ID: ${doc.getObjectId("_id")}")

    cliente.close()
    println("Conexión cerrada")
}

fun actualizarAltura() {
    //conectar con la BD
    val cliente = MongoClients.create(NOM_SRV)
    val db = cliente.getDatabase(NOM_BD)
    val colección = db.getCollection(NOM_COLECCION)

    var id_planta: Int? = null
    while (id_planta == null) {
        print("ID de la planta a actualizar: ")
        val entrada = scanner.nextLine()
        id_planta = entrada.toIntOrNull()
        if (id_planta == null) {
            println("El ID debe ser un número !!!")
        }
    }

    //comprobar si existe una planta con el id_planta proporcionado por consola
    val planta = colección.find(Filters.eq("id_planta", id_planta)).firstOrNull()
    if (planta == null) {
        println("No se encontró ninguna planta con id_planta = \"$id_planta\".")
    } else {
        // Mostrar información de la planta encontrada
        println("Planta encontrada: ${planta.getString("nombre_común")} (altura: ${planta.get("altura")} cm)")

        //pedir nueva altura
        var altura: Int? = null
        while (altura == null) {
            print("Nueva altura (en cm): ")
            val entrada = scanner.nextLine()
            altura = entrada.toIntOrNull()
            if (altura == null) {
                println("La altura debe ser un número !!!")
            }
        }
    }
}

```

```

}

// Actualizar el documento
val result = colección.updateOne(
    Filters.eq("id_planta", id_planta),
    Document("\$set", Document("altura", altura))
)

if (result.modifiedCount > 0)
    println("Altura actualizada correctamente (${result.modifiedCount} documento modificado).")
else
    println("No se modificó ningún documento (la altura quizás ya era la misma).")
}

cliente.close()
println("Conexión cerrada.")
}

fun eliminarPlanta() {
    //conectar con la BD
    val cliente = MongoClients.create(NOM_SRV)
    val db = cliente.getDatabase(NOM_BD)
    val colección = db.getCollection(NOM_COLECCION)

    var id_planta: Int? = null
    while (id_planta == null) {
        print("ID de la planta a eliminar: ")
        val entrada = scanner.nextLine()
        id_planta = entrada.toIntOrNull()
        if (id_planta == null) {
            println("El ID debe ser un número !!!")
        }
    }

    val result = colección.deleteOne(Filters.eq("id_planta", id_planta))
    if (result.deletedCount > 0)
        println("Planta eliminada correctamente.")
    else
        println("No se encontró ninguna planta con ese nombre.")

    cliente.close()
    println("Conexión cerrada.")
}
}

```

Prueba y analiza el ejemplo 5

Prueba el código de ejemplo y verifica que funciona correctamente.

Práctica 7: Trabaja con tu BD

Amplía tu proyecto con las funciones para las operaciones CRUD y un menú con las siguientes opciones:

1. Listar todos los documentos existentes.
2. Insertar un nuevo documento (a partir de los datos introducidos por consola).
3. Actualizar la información de un documento (por ID).
4. Eliminar un documento (por ID).

Consultas avanzadas y ordenación

Comando	Descripción
<code>sort()</code>	Ordena los resultados. <code>1</code> ascendente, <code>-1</code> descendente. Ejemplo: <code>db.alumnos.find().sort({nota:-1})</code>
<code>limit()</code>	Límite el número de resultados. Ejemplo: <code>db.alumnos.find().limit(3)</code>
<code>countDocuments()</code>	Devuelve el número de documentos que cumplen un filtro. Ejemplo: <code>db.alumnos.countDocuments({nota:{\$gte:5}})</code>

Índices

Comando	Descripción
<code>createIndex({campo:1})</code>	Crea un índice ascendente. Ejemplo: <code>db.alumnos.createIndex({nombre:1})</code>
<code>getIndexes()</code>	Muestra los índices existentes. Ejemplo: <code>db.alumnos.getIndexes()</code>
<code>dropIndex("nombre_1")</code>	Elimina un índice. Ejemplo: <code>db.alumnos.dropIndex("nombre_1")</code>

Consultas avanzadas con `aggregate()`

El método `aggregate()` permite realizar **consultas complejas y procesamientos de datos** en varias etapas, similares a las funciones de **GROUP BY, JOIN o HAVING** en SQL. Cada etapa del *pipeline* (tubería) transforma los datos paso a paso. Cada etapa (stage) se representa mediante un objeto precedido por \$, que indica la operación a realizar.

Estructura básica

```
db.coleccion.aggregate([
  { <etapa1> },
  { <etapa2> },
  ...
])
```

Etapa	Descripción
<code>\$match</code>	Filtra documentos (equivalente a WHERE). Ejemplo: <code>{ \$match: { ciudad: "Valencia" } }</code>
<code>\$project</code>	Selecciona campos específicos o crea nuevos. Ejemplo: <code>{ \$project: { _id:0, nombre:1, nota:1 } }</code>
<code>\$sort</code>	Ordena los resultados. Ejemplo: <code>{ \$sort: { nota: -1 } }</code>
<code>\$limit</code>	Limita el número de resultados. Ejemplo: <code>{ \$limit: 5 }</code>
<code>\$skip</code>	Omite un número de documentos. Ejemplo: <code>{ \$skip: 10 }</code>
<code>\$group</code>	Agrupa los documentos por un campo y calcula valores agregados (como COUNT, SUM, AVG). Ejemplo: <code>{ \$group: { _id: "\$curso", media: { \$avg: "\$nota" } } }</code>
<code>\$count</code>	Devuelve el número total de documentos resultantes. Ejemplo: <code>{ \$count: "total" }</code>
<code>\$lookup</code>	Realiza una unión entre colecciones (similar a JOIN). Ejemplo: <code>{ \$lookup: { from: "profesores", localField: "idProfesor", foreignField: "_id", as: "infoProfesor" } }</code>
<code>\$unwind</code>	Descompone arrays en múltiples documentos. Ejemplo: <code>{ \$unwind: "\$aficiones" }</code>

Ejemplo 6: Consultas avanzadas y agregaciones

El siguiente ejemplo realiza lo siguiente:

1. Usa `aggregate()` para calcular la altura media de las plantas.
2. Agrupa por tipo de planta con `$group` y ordena los resultados.
3. Limita la salida a los tres resultados más altos con `$limit`.

```
// Calcular altura media de todas las plantas
db.plantas.aggregate([
  { $group: { _id: null, alturaMedia: { $avg: "$altura" } } }
])

// Agrupar por tipo y calcular media, ordenar descendente
db.plantas.aggregate([
  { $match: { tipo: { $exists: true } } },
  { $group: { _id: "$tipo", mediaAltura: { $avg: "$altura" }, cantidad: { $sum: 1 } } },
  { $sort: { mediaAltura: -1 } }
])

// Obtener los 3 más altos
db.plantas.aggregate([
  { $sort: { altura: -1 } },
  { $limit: 3 },
  { $project: { _id:0, nombre_comun:1, altura:1 } }
])
```

Salida esperada:

```
// Resultado del primer aggregate
{ "_id" : null, "alturaMedia" : 250.25 }

// Resultado del group
{ "_id" : "árbol", "mediaAltura" : 540, "cantidad" : 1 }

// Resultado del limit
{ "nombre_comun" : "Olivo", "altura" : 800 }
{ "nombre_comun" : "Pino", "altura" : 330 }
{ "nombre_comun" : "Cactus", "altura" : 130 }
```



Prueba y analiza el ejemplo 6

Prueba el código de ejemplo y verifica que funciona correctamente.



Práctica 8: Trabaja sobre tu BD

1. Usa `aggregate()` para realizar algún cálculo.
2. Agrupa por tipo o categoría utilizando `$group` y ordena los resultados.
3. Limita la salida a los tres resultados más altos con `$limit`.

Ejemplo 7: Consultas avanzadas en Kotlin

El siguiente ejemplo conecta a la BD `florabotanica` y realiza las siguientes operaciones:

1. Implementa consultas utilizando filtros con `Filters.eq`, `Filters.gt`, etc.
2. Muestra solo los nombres de las plantas con `Projections.include`.
3. Realiza una agregación que calcule la media de alturas.

```
import com.mongodb.client.model.Projections
import com.mongodb.client.model.Filters

fun variasOperaciones() {
    val client = MongoClients.create(NOM_SRV)
    val col = client.getDatabase(NOM_BD).getCollection(NOM_COLECCION)

    println("*****Plantas que miden más de 100cm")
    // 1) Filtro: altura > 100
    col.find(Filters.gt("altura", 100)).forEach { println(it.toJson()) }

    println("*****Nombre común de todas las plantas")
    // 2) Proyección: solo nombre_común
    col.find().projection(Projections.include("nombre_común")).forEach { println(it.toJson()) }

    println("*****Altura media de todas las plantas")
```

```
// 3) Agregación: media de altura
    val pipeline = listOf(
        Document("\$group", Document("_id", null).append("alturaMedia", Document("\$avg", "\$altura")))
    )
    val aggCursor = col.aggregate(pipeline).iterator()
    aggCursor.use {
        while (it.hasNext()) println(it.next().toJson())
    }
    client.close()
}
```

Prueba y analiza el ejemplo 7

Prueba el código de ejemplo y verifica que funciona correctamente.

Práctica 9: Trabaja con tu BD

1. Implementa consultas utilizando filtros con `Filters.eq`, `Filters.gt`, etc.
2. Muestra solo algunos datos con `Projections.include`.
3. Realiza una agregación que realice algún cálculo sobre tus datos.

Entrega 1

Realiza lo siguiente:

1. Exporta tu BD a un archivo `.json` a la carpeta `resources` (Puedes consultar el apartado `Exportar / Importar la BD con Kotlin` al final de este documento).
2. Entrega en Aules la carpeta `main` de tu proyecto comprimida en formato `.zip`

IMPORTANTE: El proyecto no debe contener código que no se utilice, ni restos de pruebas de los ejemplos y no debe estar separado por prácticas. Debe ser un proyecto totalmente funcional.

Otras formas de trabajar con BD MongoDB

Además de instalar el servidor en local en nuestro ordenador, podemos utilizar el laboratorio de AWS. Tienes las instrucciones en la guía [Instalación y administración de MongoDB](#).

También podemos trabajar directamente del servidor online [Atlas](#) que proporciona MongoDB.

Por último se puede trabajar en local sin necesidad de servidor aunque, en este caso, la información no se guarda en disco sino en memoria y, por tanto, al cerrar la aplicación los datos se borran. Esto no es problema si se exportan a un fichero json antes de cerrar el programa y se importan al iniciararlo. Veamos un ejemplo:

Ejemplo 8: Trabajando MongoDB en local (en memoria)

Partimos del fichero `json` de la Entrega 1. A continuación se muestra la información de la colección `plantas` de la BD `florabotanica`:

```
[
  {
    "_id": {
      "$oid": "69160748005c40ac579dc29d"
    },
    "id_planta": 1,
    "nombre_comun": "Aloe",
    "nombre_cientifico": "Aloe barbadensis miller",
    "altura": 60
  },
  {
    "_id": {
      "$oid": "69160748005c40ac579dc29e"
    },
    "id_planta": 2,
    "nombre_comun": "Ficus",
    "nombre_cientifico": "Ficus benjamina",
  }
]
```

```

        "altura": 330
    },
    {
        "_id": {
            "$oid": "69160748005c40ac579dc29f"
        },
        "id_planta": 3,
        "nombre_comun": "Romero",
        "nombre_cientifico": "Rosmarinus officinalis",
        "altura": 120
    },
    {
        "_id": {
            "$oid": "69160928005c40ac579dc2a0"
        },
        "id_planta": 4,
        "nombre_comun": "Lavanda",
        "nombre_cientifico": "Lavandula angustifolia",
        "altura": 50
    },
    {
        "_id": {
            "$oid": "69160928005c40ac579dc2a1"
        },
        "id_planta": 5,
        "nombre_comun": "Clavel",
        "nombre_cientifico": "Dianthus caryophyllus",
        "altura": 60
    }
]
}

```

El código Kotlin para trabajar con MongoDB en memoria de forma local es el siguiente:

build.gradle.kts

```

dependencies {
    implementation("org.mongodb:mongodb-driver-sync:4.10.2")
    implementation("de.bwaldvogel:mongo-java-server:1.45.0")
    implementation("org.json:json:20231013")

    // Backend de logging para SLF4J
    //implementation("ch.qos.logback:logback-classic:1.5.6")

    //desactivar logs en consola
    implementation("org.slf4j:slf4j-nop:2.0.12")
}

```

Main.kt

```

import java.util.Scanner
import java.io.File
import org.bson.Document
import org.json.JSONArray
import org.bson.json.JsonWriterSettings

import de.bwaldvogel.mongo.MongoServer
import de.bwaldvogel.mongo.backend.memory.MemoryBackend

import com.mongodb.client.MongoClients
import com.mongodb.client.MongoClient
import com.mongodb.client.MongoCollection
import com.mongodb.client.model.Filters
import com.mongodb.client.model.Projections
import com.mongodb.client.model.Aggregates

//variables globales definidas sin inicializar
lateinit var servidor: MongoServer
lateinit var cliente: MongoClient
lateinit var uri: String
lateinit var colecciónPlantas: MongoCollection<Document>

//BD y colección con la que se trabajará
const val NOM_BD = "florabotanica"
const val NOM_COLECCION = "plantas"

// Función para conectar a la BD
fun conectarBD() {
    servidor = MongoServer(MemoryBackend())
    val address = servidor.bind()
    uri = "mongodb://{$address.hostName}:{$address.port}"

    cliente = MongoClients.create(uri)
    colecciónPlantas = cliente.getDatabase(NOM_BD).getCollection(NOM_COLECCION)

    println("Servidor MongoDB en memoria iniciado en $uri")
}

// Función para desconectar a la BD
fun desconectarBD() {
    cliente.close()
    servidor.shutdown()
}

```

```

    println("Servidor MongoDB en memoria finalizado")
}

fun main() {
    conectarBD()
    importarBD("src/main/resources/florabotanica_plantas.json", colecciónPlantas)

    menu()

    exportarBD(colecciónPlantas, "src/main/resources/florabotanica_plantas.json")
    desconectarBD()
}

// *****
// ***** MENÚ *****
// *****

fun menu(){
    //Llamada a listar todas las plantas de la BD
    mostrarPlantas()
}

fun mostrarPlantas() {
    println();
    println("**** Listado de plantas:")
    colecciónPlantas.find().forEach { doc ->
        val id = doc.getInteger("id_planta")
        val nombre_común = doc.getString("nombre_común")
        val nombre_científico = doc.getString("nombre_científico")
        val altura = doc.getInteger("altura")
        println("[${id}] ${nombre_común} (${nombre_científico}): ${altura} cm")
    }
}

```

Hay que tener en cuenta que cuando se trabaja con un servidor en memoria no podemos abrir y cerrar la conexión a la BD en cada operación ya que todo está en la RAM y no en disco. Cada vez que se cierra la conexión y el servidor, la base de datos desaparece y por esta razón la abrimos al iniciar el programa y la cerramos al finalizarlo. También importamos la información y la exportamos para tener siempre una copia en formato `json`.

Prueba y analiza el ejemplo 8

Prueba el código del ejemplo y verifica que funciona correctamente.

Práctica 10: Trabaja con tu BD

1. Crea un nuevo proyecto kotlin.
2. Copia el fichero `json` que exportaste en la práctica anterior a la carpeta `resources`.
3. Añade el código de tu anterior proyecto modificando lo que sea necesario para que funcionen todas las opciones de la práctica anterior pero trabajando la BD en memoria.
4. Comprueba que al salir del programa el fichero `json` contiene la información actualizada.

Trabajando con más de una colección

En este punto vamos a profundizar en la utilización de `aggregate()` para poder realizar **consultas complejas y procesamientos de datos**. Para ello utilizaremos una lista de etapas (*stages*) que MongoDB ejecutará **en orden** para transformar, combinar o procesar documentos de una colección. Esa lista la guardaremos en una **tubería de pasos** (*pipeline*), donde la salida de un paso es la entrada del siguiente.

Ya hemos visto que listar el contenido de una colección es muy fácil utilizando `find`, pero si queremos realizar consultas que obtengan datos de varias colecciones hay que realizar operaciones similares al `JOIN` de SQL.

Para los ejemplos siguientes añadiremos una nueva colección (facturas) a nuestra BD. A continuación se muestra su estructura e información inicial:

Campo	Tipo
fecha	String (formato YYYY-MM-DD)
id_factura	Integer
id_planta	Integer
precio	Integer
cantidad	Integer

```
[
{
  "fecha": "2025-11-28",
  "id_factura": 1,
  "id_planta": 1,
  "precio": 13,
  "cantidad": 3
},
{
  "fecha": "2025-11-28",
  "id_factura": 1,
  "id_planta": 3,
  "precio": 7,
  "cantidad": 2
},
{
  "fecha": "2025-11-28",
  "id_factura": 1,
  "id_planta": 5,
  "precio": 5,
  "cantidad": 1
},
{
  "fecha": "2025-11-28",
  "id_factura": 2,
  "id_planta": 2,
  "precio": 35,
  "cantidad": 1
},
{
  "fecha": "2025-11-28",
  "id_factura": 2,
  "id_planta": 4,
  "precio": 9,
  "cantidad": 2
},
{
  "fecha": "2025-11-29",
  "id_factura": 3,
  "id_planta": 1,
  "precio": 13,
  "cantidad": 1
},
{
  "fecha": "2025-11-29",
  "id_factura": 3,
  "id_planta": 3,
  "precio": 7,
  "cantidad": 3
},
{
  "fecha": "2025-11-29",
  "id_factura": 4,
  "id_planta": 2,
  "precio": 35,
  "cantidad": 2
},
{
  "fecha": "2025-11-29",
  "id_factura": 4,
  "id_planta": 5,
  "precio": 5,
  "cantidad": 4
}
]
```

Para realizar el JOIN entre las dos colecciones (plantas y facturas) utilizaremos **lookup y unwind**.

`lookup` añade un nuevo campo que contiene un array con los documentos completos de otra colección cuyo campo coincide con el del documento actual. Incluso si solo encuentra un documento, el resultado sigue siendo un array con un único elemento.

Partimos del primer documento de la colección `facturas`:

```
{
  "fecha": "2025-11-28",
  "id_factura": 1,
  "id_planta": 1,
  "precio": 13,
  "cantidad": 3
}
```

Utilizamos `lookup` para buscar en la colección **plantas** todos los documentos cuyo campo `id_planta` coincida con el `id_planta` de la factura y gurdarlos en un nuevo campo llamado `planta`. El código es el siguiente:

```
Document("\$lookup", Document()
.append("from", "plantas")
.append("localField", "id_planta")
.append("foreignField", "id_planta")
.append("as", "planta")
)
```

El resultado (equivalente a un **JOIN** en SQL) es un campo llamado `planta` añadido al documento:

```
{
  "fecha": "2025-11-28",
  "id_factura": 1,
  "id_planta": 1,
  "precio": 13,
  "cantidad": 3
  "planta": [
    {
      "nombre_comun": "Aloe",
      "nombre_cientifico": "Aloe barbadensis miller",
      "altura": 60,
      "id_planta": 1
    }
  ]
}
```

Para poder leer la información hemos de convertir el resultado del `lookup` (array) en un objeto normal. Eso es lo que hace `unwind`. Si partimos del array que se ha creado con `lookup`:

```
"planta": [
  { nombre_comun: "Aloe", ... }
]
```

Después de aplicar `unwind` el documento de planta ya no es un `array` y queda como un objeto normal para poder leer sus campos.

```
"planta": {
  nombre_comun: "Aloe",
  ...
}
```

En este caso, el pipeline es una secuencia de dos pasos:

1. `lookup` que junta facturas con plantas (como un JOIN).
2. `unwind` que convierte el resultado del `lookup` (array) en un objeto normal.

Ejemplo 9: Mostrar listado de factura con el nombre de la planta

Este ejemplo utiliza el `pipeline` explicado anteriormente con la secuencia `lookup` y `unwind` para mostrar el nombre de la planta al listar los documentos de la colección `facturas`.

```
val pipeline = listOf(
  Document("\$lookup", Document()
    .append("from", "plantas")
    .append("localField", "id_planta")
    .append("foreignField", "id_planta")
    .append("as", "planta")
  ),
  Document("\$unwind", "\$planta")
)
```

```

colecciónFacturas.aggregate(pipeline).forEach { doc ->
    val idFactura = doc.getInteger("id_factura")
    val fecha = doc.getString("fecha")
    val idPlanta = doc.getInteger("id_planta")
    val cantidad = doc.getInteger("cantidad")
    val precio = doc.getInteger("precio")

    val planta = doc["planta"] as Document
    val nombreComún = planta.getString("nombre_común")

    println("[idFactura] ($fecha): $nombreComún (id $idPlanta) - $cantidad uds. $precio €")
}
}

```

Prueba y analiza el ejemplo 9

Prueba el código del ejemplo y verifica que funciona correctamente.

Ejemplo 10: Mostrar datos de una factura

En este ejemplo se pide un número de factura por consola y se muestran sus datos.

```

fun mostrarFactura() {
    val idFactura = pedirEntero("ID de la factura: ")

    // Obtener la fecha de la factura y verificar que la factura indicada existe
    val facturaDoc = colecciónFacturas
        .find(Document("id_factura", idFactura))
        .first()

    if (facturaDoc == null) {
        println("No existe ninguna factura con ID $idFactura")
        return
    }

    val fecha = facturaDoc["fecha"] as String

    // Crear un pipeline de agregación para obtener las líneas de la factura con datos de la planta
    val pipeline = listOf(
        Document("\$match", Document("id_factura", idFactura)),
        Document("\$lookup", Document()
            .append("from", "plantas")
            .append("localField", "id_planta")
            .append("foreignField", "id_planta")
            .append("as", "planta")),
        Document("\$unwind", "\$planta"),
        Document("\$project", Document()
            .append("nombre_planta", "\$planta.nombre_común")
            .append("cantidad", 1)
            .append("precio", 1)
            .append("subtotal", Document("\$multiply", listOf("\$precio", "\$cantidad"))))
    )
}

// Ejecutar la agregación para obtener la lista de líneas
val líneas = colecciónFacturas.aggregate(pipeline).toList()

if (líneas.isEmpty()) {
    println("No se encontraron líneas para la factura $idFactura")
    return
}

// Encabezado de la factura
println("====")
println("Factura ID: $idFactura")
println("Fecha: $fecha")
println("-----")
println(String.format("%-15s %-10s %-10s %-12s", "Planta", "Cantidad", "Precio", "Subtotal"))
println("-----")

var totalFactura = 0.0

// Iterar sobre las líneas de la factura
líneas.forEach { linea ->
    val nombre = linea["nombre_planta"] as String
    val cantidad = linea["cantidad"] as Int
    val precio = linea["precio"] as Int
    val subtotal = (linea["subtotal"] as Number).toDouble()

    totalFactura += subtotal

    println(String.format("%-15s %-10d %-10s %-12s",
        nombre, cantidad, precio, subtotal
    ))
}
}

```

```

var totalIVA =totalFactura*0.21

// Mostrar pie de factura con totales
println("-----")
println(String.format("%-15s %-10s %-10s %-12s", "", "TOTAL:", totalFactura, ""))
println(String.format("%-15s %-10s %-10s %-12s", "", "IVA 21%:", totalIVA, ""))
println(String.format("%-15s %-10s %-10s %-12s", "", "TOTAL CON IVA:", totalFactura + totalIVA, ""))
println("=====")
}

```

Prueba y analiza el ejemplo 10

Prueba el código del ejemplo y verifica que funciona correctamente.

Práctica 11: Trabaja con tu BD

1. Añade una nueva colección a tu BD.
2. Añade al menú las operaciones CRUD.
3. Programa una función parecida a la de los ejemplos en la que tengas que realizar una consulta para extraer información de las dos colecciones de tu BD.
4. Recuerda importar y exportar la nueva colección.

Ejemplo 11: Trabajando con una tercera colección

Vamos a completar un poco más nuestra aplicación añadiendo la colección de `Clients`. Los datos iniciales son los siguientes:

```
[{
  "nombre": "Pol",
  "id_cliente": 1
},
{
  "nombre": "Ade",
  "id_cliente": 2
}]
```

Vamos a practicar realizando las siguientes consultas:

Consulta 1: Mostrar el nombre del cliente en la factura del ejemplo anterior

La cabecera de la función `mostrarFactura` del ejemplo anterior era la siguiente:

```
=====
Factura ID: 1
Fecha: 2025-11-28
```

Con la modificación queda así:

```
=====
Factura ID: 1
Fecha: 2025-11-28
Cliente: Pol
```

Para obtener el nombre del cliente hay que añadir las siguientes instrucciones:

```

val idCliente = facturaDoc.getInteger("id_cliente")

val clienteDoc = colecciónClientes
  .find(Document("id_cliente", idCliente))
  .first()

val nomCliente = clienteDoc.getString("nombre")
```

Consulta 2: Historial de compras

Para este ejemplo vamos a unir las tres colecciones para mostrar el historial de compras. El resultado (sin formatear) será el siguiente:

```
{
  "cliente": "Ade", "id_factura": 4, "fecha": "2025-11-29", "planta": "Ficus", "cantidad": 2, "precio": 35}
  {"cliente": "Ade", "id_factura": 4, "fecha": "2025-11-29", "planta": "Clavel", "cantidad": 4, "precio": 5}
  {"cliente": "Pol", "id_factura": 1, "fecha": "2025-11-28", "planta": "Aloe", "cantidad": 3, "precio": 13}
  {"cliente": "Pol", "id_factura": 1, "fecha": "2025-11-28", "planta": "Romero", "cantidad": 2, "precio": 7}
  {"cliente": "Pol", "id_factura": 1, "fecha": "2025-11-28", "planta": "Clavel", "cantidad": 1, "precio": 5}
  {"cliente": "Pol", "id_factura": 2, "fecha": "2025-11-28", "planta": "Ficus", "cantidad": 1, "precio": 35}
  {"cliente": "Pol", "id_factura": 2, "fecha": "2025-11-28", "planta": "Lavanda", "cantidad": 2, "precio": 9}
  {"cliente": "Pol", "id_factura": 3, "fecha": "2025-11-29", "planta": "Aloe", "cantidad": 1, "precio": 13}
  {"cliente": "Pol", "id_factura": 3, "fecha": "2025-11-29", "planta": "Romero", "cantidad": 3, "precio": 7}
  {"cliente": "Pol", "id_factura": 90, "fecha": "2025-12-02", "planta": "dfdfdfd", "cantidad": 2, "precio": 50}
}
```

Para obtener el nombre del cliente en cada una de las líneas de factura, habrá que hacer un nuevo **lookup** y un nuevo **unwind** con la colección de clientes, por tanto añadiremos a la 'pipeline' el siguiente código:

```
Document("\$lookup", Document()
  .append("from", "clientes")
  .append("localField", "id_cliente")
  .append("foreignField", "id_cliente")
  .append("as", "cliente"))
),
Document("\$unwind", "\$cliente"),
```

y la proyección quedará de esta forma:

```
Document("\$project", Document()
  .append("_id", 0)
  .append("id_factura", 1)
  .append("fecha", 1)
  .append("planta", "\$planta.nombre_comun")
  .append("cantidad", 1)
  .append("precio", 1)
  .append("subtotal", Document("\$multiply", listOf("\$precio", "\$cantidad")))
)
```

Consulta 3: Total gastado por cliente

Esta consulta saca como resultado una línea por cada cliente con su nombre, la suma total de todas las líneas de facturas (precio * cantidad) y el número total de líneas:

```
{"_id": "Ade", "total_gastado": 90, "lineas_compradas": 2}
{"_id": "Pol", "total_gastado": 245, "lineas_compradas": 8}
```

En este caso, los **lookup** y **unwind** serán los mismos que en la consulta anterior pero en vez de una proyección necesitamos un agrupamiento. Este es el código:

```
Document("\$group", Document()
  .append("_id", "\$cliente.nombre")
  .append("total_gastado", Document("\$sum",
    Document("\$multiply", listOf("\$precio", "\$cantidad")))
  ))
  .append("lineas_compradas", Document("\$sum", 1))
)
```

Consulta 4: Plantas más vendidas con clientes compradores

En este caso vamos a obtener la información ordenada de las plantas que se han vendido junto a los clientes que las han comprado. El listado (sin formatear) es el siguiente:

```
{"_id": "Clavel", "total_vendida": 5, "clientes": ["Ade", "Pol"]}
{"_id": "Romero", "total_vendida": 5, "clientes": ["Pol"]}
{"_id": "Aloe", "total_vendida": 4, "clientes": ["Pol"]}
{"_id": "Ficus", "total_vendida": 3, "clientes": ["Ade", "Pol"]}
{"_id": "Lavanda", "total_vendida": 2, "clientes": ["Pol"]}
{"_id": "dfdfdfd", "total_vendida": 2, "clientes": ["Pol"]}
```

Para obtener la información de esta forma el código necesario, además de los **lookup** y **unwind** es el siguiente:

```
Document("\$group", Document()
  .append("_id", "\$planta.nombre_comun")
  .append("total_vendida", Document("\$sum", "\$cantidad"))
  .append("clientes", Document("\$addToSet", "\$cliente.nombre"))
),
Document("\$sort", Document("total_vendida", -1))
```

✓ Prueba y analiza el ejemplo 11

Prueba el código de las consultas del ejemplo y verifica que todas funcionan correctamente.

⚠️ Práctica 12: Trabaja con tu BD

1. Añade una nueva colección a tu BD.
2. Añade al menú las operaciones CRUD.
3. Programa al menos tres funciones parecidas a las de los ejemplos. Formatea el resultado para que salga bonito.
4. Añade al menú de tu aplicación todas las opciones necesarias para poder ejecutar cada función.
5. Recuerda importar y exportar la nueva colección.
6. Crea un fichero `README.md` dentro de la carpeta `main`. Su contenido debe ser un manual de usuario con los siguientes apartados:
 - Descripción general.
 - Requisitos.
 - Base de datos.
 - Cómo ejecutar.
 - Opciones del programa y ejemplos de uso (salida por consola).
 - Notas importantes (si hay algo que destacar).

⚡ Entrega 2

Realiza lo siguiente:

1. Asegúrate que tu aplicación exporta correctamente todas las colecciones de tu BD (cada una a un archivo `.json`) y que se guardan en la carpeta `resources` (consulta el apartado `Exportar / Importar la BD con Kotlin` al final de este documento).
2. Entrega en Aules la carpeta `main` de tu proyecto comprimida en formato `.zip`

IMPORTANTE: El proyecto no debe contener código que no se utilice, ni restos de pruebas de los ejemplos y no debe estar separado por prácticas. Debe ser un proyecto totalmente funcional.

Resumen

Categoría	Comandos clave
Base de datos	<code>show dbs</code> , <code>use</code> , <code>db.getName()</code> , <code>db.dropDatabase()</code>
Colecciones	<code>show collections</code> , <code>db.createCollection()</code> , <code>db.coleccion.drop()</code>
Inserción	<code>db.coleccion.insertOne()</code> , <code>db.coleccion.insertMany()</code>
Consulta	<code>db.coleccion.find()</code> , <code>db.coleccion.findOne()</code> , <code>.sort()</code> , <code>.limit()</code>
Actualización	<code>db.coleccion.updateOne()</code> , <code>db.coleccion.updateMany()</code> , <code>\$set</code>
Eliminación	<code>db.coleccion.deleteOne()</code> , <code>db.coleccion.deleteMany()</code>
Índices	<code>db.coleccion.createIndex()</code> , <code>db.coleccion.getIndexes()</code> , <code>db.coleccion.dropIndex()</code>
Estadísticas	<code>db.stats()</code> , <code>db.coleccion.stats()</code> , <code>db.version()</code>

Errores comunes y cómo resolverlos

1) Error: `Error opening socket / unable to connect` - **Causa:** el servidor MongoDB no está en ejecución o la URI es incorrecta. - **Solución:** arrancar el servicio (`sudo systemctl start mongod` en Linux) o comprobar `mongod` en Windows (servicios) y verificar la URI `mongodb://localhost:27017`.

2) Authentication failed - **Causa:** autenticación habilitada y credenciales no proporcionadas. - **Solución:** crear un usuario en `admin` con `db.createUser()` o usar una URI con usuario/contraseña: `mongodb://user:pwd@localhost:27017`.

3) NamespaceNotFound / colección no encontrada - **Causa:** la colección no existe (no se creó o no tiene documentos). - **Solución:** insertar un documento o crear la colección explícitamente con `db.createCollection("plantas")`.

4) BSONTypeError o problemas de tipo al recuperar datos - **Causa:** tipos inconsistentes (por ejemplo, altura a veces string, a veces número). - **Solución:** normalizar datos o validar antes de insertar; usar `$convert` en agregaciones o transformar en la app.

5) Problemas con dependencias en Kotlin - **Causa:** dependencia no encontrada o versión incompatible. - **Solución:** comprobar `build.gradle.kts` y usar `mavenCentral()`; actualizar la versión del driver.

6) No primary found en replicación o clúster - **Causa:** intentando escribir en un conjunto de réplicas sin primario. - **Solución:** comprobar estado del replicaset (`rs.status()`) o arrancar una instancia standalone para prácticas locales.

Exportar / Importar la BD con Kotlin

Desde Kotlin podemos exportar nuestra BD a un archivo `.json` y también podemos importar un archivo `.json` a nuestra BD. Para ello hay que añadir la siguiente dependencia en el archivo `build.gradle.kts`.

```
implementation("org.json:json:20231013")
```

A continuación se muestra el código que exporta la BD a un archivo `.json` (actualizado para tomar como parámetros la ruta del json y el nombre de la colección).

```
import com.mongodb.client.MongoClients
import org.bson.json.JsonWriterSettings
import java.io.File

fun exportarBD(coleccion: MongoCollection<Document>, rutaJSON: String) {
    val settings = JsonWriterSettings.builder().indent(true).build()
    val file = File(rutaJSON)
    file.printWriter().use { out ->
        out.println("[")
        val cursor = coleccion.find().iterator()
        var first = true
        while (cursor.hasNext()) {
            if (!first) out.println(",")
            val doc = cursor.next()
            out.print(doc.toJson(settings))
            first = false
        }
        out.println("]")
        cursor.close()
    }
    println("Exportación de ${coleccion.namespace.collectionName} completada")
}
```

A continuación se muestra el código que importa la BD desde un archivo `.json` (actualizado para tomar como parámetros la ruta del json y el nombre de la colección).

```
import com.mongodb.client.MongoClients
import org.bson.Document
import org.json.JSONArray
import java.io.File

fun importarBD(rutaJSON: String, coleccion: MongoCollection<Document>) {
    println("Iniciando importación de datos desde JSON...")
    val jsonFile = File(rutaJSON)
    if (!jsonFile.exists()) {
        println("No se encontró el archivo JSON a importar")
        return
    }
    // Leer JSON del archivo
    val jsonText = try {
        jsonFile.readText()
    } catch (e: Exception) {
        println("Error leyendo el archivo JSON: ${e.message}")
        return
    }
```

```

val array = try {
    JSONArray(jsonText)
} catch (e: Exception) {
    println("Error al parsear JSON: ${e.message}")
    return
}

// Convertir JSON a Document y eliminar _id si existe
val documentos = mutableListOf<Document>()
for (i in 0 until array.length()) {
    val doc = Document.parse(array.getJSONObject(i).toString())
    doc.remove("_id") // <-- eliminar _id para que MongoDB genere uno nuevo
    documentos.add(doc)
}

if (documentos.isEmpty()) {
    println("El archivo JSON está vacío")
    return
}

val db = cliente.getDatabase(NOM_BD)

val nombreColeccion = colección.namespace.collectionName

// Borrar colección si existe
if (db.listCollectionNames().contains(nombreColeccion)) {
    db.getCollection(nombreColeccion).drop()
    println("Colección '$nombreColeccion' eliminada antes de importar.")
}

// Insertar documentos
try {
    colección.insertMany(documentos)
    println("Importación completada: ${documentos.size} documentos de $nombreColección.")
} catch (e: Exception) {
    println("Error importando documentos: ${e.message}")
}
}

```

Autoría

Obra realizada por Begoña Paterna Lluch basada en materiales desarrollados por Alicia Salvador Contreras. Publicada bajo licencia [Creative Commons Atribución/Reconocimiento-CompartirIgual 4.0 Internacional](#)

2.3 Unidad 4. Componentes (Spring Framework)

Revisiones

Revisión	Fecha	Descripción
1.0	30-12-2025	Adaptación de los materiales a markdown

2.3.1 4.1. Introducción

Spring es un framework de código abierto para crear aplicaciones en Java o Kotlin de forma más fácil, rápida y ordenada. Facilita el trabajo de crear objetos, conectar clases, preparar la base de datos y configurar servidores.

Spring se basa principalmente en:

- **Inversión de Control (IoC):** Se encarga de crear y gestionar los objetos de la aplicación.
- **Inyección de Dependencias (DI):** Coloca los objetos donde hacen falta automáticamente.

Además tiene tres pilares:

1. Autoconfiguración (Spring Boot): prepara el proyecto por ti

- Servidor web.
- Conexión a base de datos.
- Estructura de proyecto.
- Dependencias necesarias.

2. Starters: paquetes listos para usar según lo que quieras hacer

Starter	Descripción
spring-boot-starter-web	para rutas y controladores
spring-boot-starter-thymeleaf	para páginas HTML
spring-boot-starter-data-jpa	para BD y CRUD

3. Anotaciones: indican qué hace cada clase

Las anotaciones son etiquetas especiales que se colocan encima de clases, funciones o atributos para decirle a Spring cómo debe comportarse con ese código. Las anotaciones son, por tanto, la forma en la que Spring entiende la aplicación. Spring tiene muchísimas anotaciones, porque es un framework muy grande y sirve para muchos tipos de proyectos (web MVC, microservicios, seguridad, batch, mensajería, etc.).

En nuestro caso, como vamos a trabajar únicamente con Spring Boot, API REST, vistas HTML y JPA, no es necesario aprender todas las anotaciones que ofrece Spring. Basta con conocer un conjunto reducido de anotaciones básicas, suficientes para desarrollar un backend completo y funcional.

En las siguientes tablas se recogen las anotaciones más importantes que utilizaremos a lo largo del tema (para API REST/vistas HTML + JPA) (a medida que avancemos, irán apareciendo otras anotaciones adicionales que se introducirán solo cuando sean necesarias para la aplicación):

- Anotaciones de arranque de la app

Anotación	Dónde se usa	Para qué sirve
@SpringBootApplication	Clase principal	Marca la clase de arranque de la aplicación Spring Boot y activa la auto- configuración y el escaneo de componentes

- Anotaciones API REST

Anotación	Dónde se usa	Para qué sirve
@RestController	Clase	Indica que la clase es un controlador REST y que los métodos devuelven directamente datos (normalmente JSON).
@RequestMapping	Clase o método	Define la ruta base o una ruta concreta para acceder a un recurso
@GetMapping	Método	Atiende peticiones HTTP GET (lectura de datos)
@PostMapping	Método	Atiende peticiones HTTP POST (creación de datos)
@PutMapping	Método	Atiende peticiones HTTP PUT (actualización de datos)
@DeleteMapping	Método	Atiende peticiones HTTP DELETE (eliminación de datos)
@RequestBody	Parámetro	Permite recibir datos enviados en el cuerpo de la petición (JSON)
@PathVariable	Parámetro	Permite recoger valores de la URL (por ejemplo, un identificador)

- Anotaciones MVC (vistas)

Anotación	Dónde se usa	Para qué sirve
@Controller	Clase	Marca una clase como controlador MVC tradicional, devolviendo vistas (HTML con Thymeleaf)

- Anotaciones de lógica de negocio

Anotación	Dónde se usa	Para qué sirve
@Service	Clase	Marca una clase como servicio, donde se implementa la lógica de negocio
@Autowired	Atributo o constructor	Inyecta automáticamente una dependencia gestionada por Spring

- Anotaciones JPA / Base de datos

Anotación	Dónde se usa	Para qué sirve
@Entity	Clase	Indica que la clase representa una tabla de la base de datos
@Table	Clase	Define el nombre de la tabla asociada a la entidad
@Id	Atributo	Marca el atributo como clave primaria
@GeneratedValue	Atributo	Indica que el valor de la clave primaria se genera automáticamente
@Column	Atributo	Configura una columna de la tabla (nombre, restricciones, unicidad, etc.)
@OneToMany	Atributo	Define una relación uno-a-muchos entre entidades
@ManyToOne	Atributo	Define una relación muchos-a-uno entre entidades
@JoinColumn	Atributo	Especifica la columna usada como clave foránea en una relación

- Anotaciones de acceso a datos

Anotación	Dónde se usa	Para qué sirve
@Repository	Clase o interfaz	Indica que la clase o interfaz se encarga del acceso a datos y de la gestión de excepciones de base de datos

Los componentes principales de Spring Framework son:

Componente	Descripción
Spring Core	El núcleo del framework, encargado de la inyección de dependencias
Spring Boot	Facilita la creación de aplicaciones basadas en Spring con una configuración mínima
Spring MVC	Permite el desarrollo de aplicaciones web utilizando el patrón Modelo-Vista-Controlador
Spring Data	Simplifica el acceso a datos con soporte para JPA, MongoDB, Redis, entre otros
Spring Security	Proporciona herramientas para implementar seguridad en aplicaciones
Spring Cloud	Ayuda en la construcción de aplicaciones distribuidas y microservicios

2.3.2 4.2. Spring Boot

Para crear una aplicación se necesita crear el proyecto, desarrollar la aplicación y desplegarla en un servidor. **Spring Boot** simplifica las tareas de crear el proyecto y desplegar la aplicación ya que:

- Configura todo automáticamente.
- Trae un servidor web incorporado (permite crear aplicaciones que se ejecutan de forma independiente sin necesidad de un servidor web externo).
- Evita escribir XML.
- Permite arrancar una app con un botón.
- Usa starters (dependencias ya preparadas).
- Permite crear proyectos en segundos.

Para crear un proyecto **Spring Boot** Maven/Gradle con las dependencias necesarias tenemos dos opciones:

- Crear un proyecto Spring Boot utilizando la herramienta Spring Initializr desde la url <https://start.spring.io/> la cual genera un proyecto base con la estructura de una aplicación Spring Boot en un archivo .zip que podemos abrir directamente desde un IDE.
- Crear un proyecto Spring Boot utilizando un IDE que tenga instalados los plugins necesarios. En el caso de IntelliJ solamente es posible utilizar el plugin de Spring en la versión Ultimate.

Una vez creado el proyecto tendremos las configuraciones y dependencias en los archivos siguientes:

- **application.properties:** configuración de aspectos como las conexiones a base de datos o el puerto por donde acceder a nuestra aplicación.
- **pom.xml:** dependencias necesarias para que la aplicación funcione.

Dependencia Spring Web

- Se utiliza para desarrollar aplicaciones web, ya sea basadas en REST o tradicionales con HTML dinámico.
- Incluye un servidor web embebido (por defecto, Tomcat) para ejecutar la aplicación sin necesidad de configurarlo manualmente.
- Facilita el manejo de rutas HTTP (GET, POST, PUT, DELETE, etc.) y parámetros de solicitud a través de métodos en los controladores.
- Usa la biblioteca Jackson (incluida por defecto) para convertir automáticamente objetos Kotlin/Java a JSON y viceversa.
- Ofrece herramientas para manejar errores y excepciones de forma global mediante @ControllerAdvice o controladores personalizados.

Ejemplo 1: Aplicación Spring Boot con Spring Web

A continuación se describen los pasos para crear una aplicación que saluda al usuario utilizando Spring Web.

PASO 1: Crear el proyecto

Accedemos a Spring Initializr desde la url <https://start.spring.io/>, indicamos el nombre de la aplicación y añadimos la dependencia **Spring Web** (el resto de opciones las podemos dejar como se ve en la imagen). Por último hacemos clic en el botón GENERATE. Esto hará que se cree el proyecto y se descargue en un archivo .zip.

The screenshot shows the Spring Initializr web interface with the following configuration:

- Project:** Maven (selected)
- Language:** Java (selected)
- Spring Boot:** 4.0.1 (selected)
- Dependencies:** Spring Web (selected)
- Project Metadata:**
 - Group: com.example
 - Artifact: saludo
 - Name: saludo
 - Description: Demo project for Spring Boot
 - Package name: com.example.saludo
 - Packaging: Jar (selected)
 - Configuration: Properties (selected)
 - Java: 21 (selected)
- Buttons at the bottom:**
 - GENERATE (highlighted with a red box)
 - EXPLORE (CTRL + SPACE)
 - ...

PASO 2: Abrir el proyecto y ejecutarlo

Descomprimimos el archivo obtenido en el paso anterior y lo abrimos con IntelliJ. Vemos que, además de los archivos **application.properties** y **pom.xml** se ha creado automáticamente la clase **SaludoApplication** (con la anotación **@SpringBootApplication**) y la función de extensión **runApplication** que sirve para lanzar la aplicación.

Por tanto deberemos ejecutar la aplicación usando la clase `SaludoApplication.kt` como clase principal. Al ejecutar la aplicación veremos por Consola la salida de los mensajes de registro de Spring.

```

   _    _ 
  / \ / - - - - ( ) - - - - \ \ \ \ \
( ()\_\_ | ' - | ' - | ' \ / ' | \ \ \ \
\ \ \_\_)| | - | | | | | | | | | ) ) )
' | - | - - | - | - | - \ , | / / /
=====|_|=====
:: Spring Boot ::          (v4.0.1)

2025-12-30T22:21:45.613+01:00 INFO 8680 --- [saludo] [
main] com.example.saludo.SaludoApplicationKt : Starting SaludoApplicationKt using Java 21.0.9 with PID 8680
2025-12-30T22:21:45.624+01:00 INFO 8680 --- [saludo] [
main] com.example.saludo.SaludoApplicationKt : No active profile set, falling back to 1 default profile: "d
2025-12-30T22:21:47.599+01:00 INFO 8680 --- [saludo] [
main] o.s.boot.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http)
2025-12-30T22:21:47.624+01:00 INFO 8680 --- [saludo] [
main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2025-12-30T22:21:47.625+01:00 INFO 8680 --- [saludo] [
main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/11.0.15]
2025-12-30T22:21:47.735+01:00 INFO 8680 --- [saludo] [
main] b.w.c.s.WebApplicationContextInitializer : Root WebApplicationContext: initialization completed in 1944
2025-12-30T22:21:49.779+01:00 INFO 8680 --- [saludo] [
main] o.s.boot.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path '/'
2025-12-30T22:21:49.805+01:00 INFO 8680 --- [saludo] [
main] com.example.saludo.SaludoApplicationKt : Started SaludoApplicationKt in 5.209 seconds (process runnin

```

Si el puerto 8080 está ocupado aparecerá un mensaje diciendo que no se puede iniciar el servidor Tomcat. Puedes cambiar el puerto, por ejemplo al 8888, añadiendo la siguiente línea en el archivo `application.properties` (que se encuentra en la carpeta resources del proyecto):

```
server.port=8888
```

PASO 3: Añadir el código para saludar

Añadimos a la clase principal `SaludoApplication` la función `sayHello()` con el código necesario para que nuestra aplicación envíe un saludo:

```
@SpringBootApplication
@RestController
class SaludoApplication{
    @GetMapping("/hello")
```

```

fun sayHello(
    @RequestParam(value = "myName", defaultValue = "World") name: String): String
{
    return "Hello $name!"
}
}

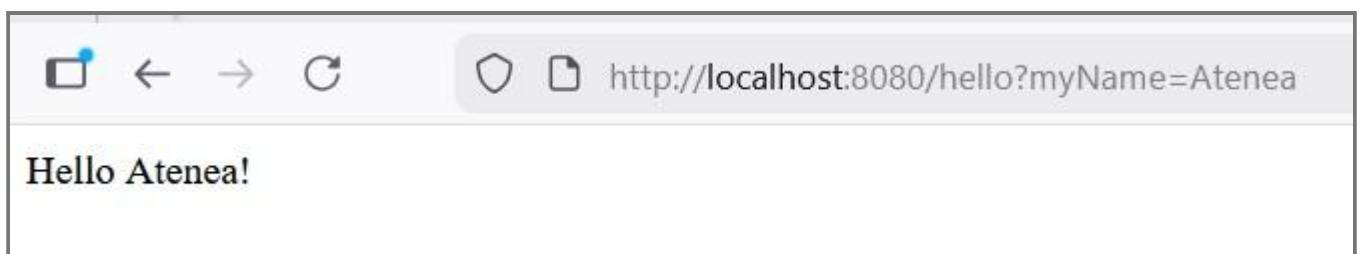
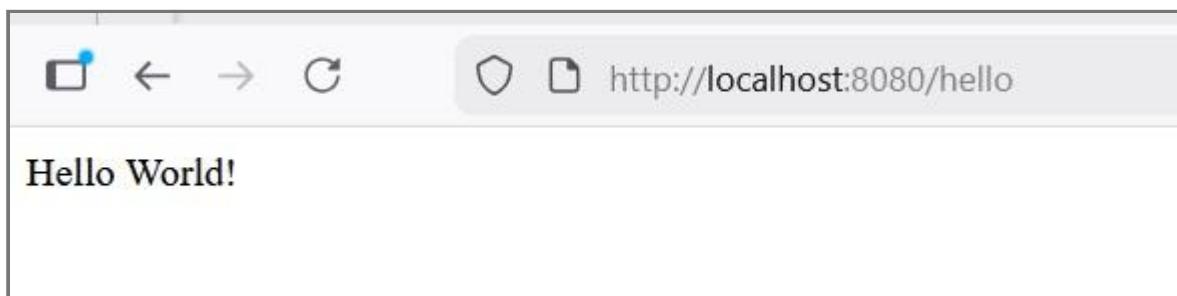
```

Como puedes ver, se han incluido anotaciones e importaciones, a continuación se explica cada una de ellas:

- **@RestController**: se utiliza para que Spring reconozca la clase como un controlador que maneja solicitudes HTTP. Combina:
- **@Controller**: Define la clase como un controlador web.
- **@ResponseBody**: Indica que los métodos devolverán directamente el cuerpo de la respuesta (en este caso, texto plano en lugar de una vista HTML).
- **@GetMapping("/hello")**: Es una anotación de Spring que indica que este método debe manejar las solicitudes HTTP GET que lleguen a la URL /hello.
- Enlaza la URL /hello con el método sayHello.
- Cada vez que se acceda a la ruta <http://localhost:8080/hello> (asumiendo el puerto predeterminado 8080) en un navegador con un método GET, Spring ejecutará el método sayHello.
- **@RequestParam**: se usa para extraer un parámetro de la consulta (query parameter) enviado en la URL.
- El método espera un parámetro de consulta llamado `myName`.
- Si el cliente no incluye myName en la solicitud, el valor predeterminado será "World", gracias a `defaultValue = "World"`.

PASO 4: Volvemos a ejecutar la aplicación

Ejecutamos la aplicación para levantar el servidor y abrimos la dirección <http://localhost:8080/hello> o <http://localhost:8080/hello?myName=Atenea> en el navegador web. La aplicación responde con Hello World! o con Hello Atenea! (que es el nombre pasado como parámetro):



PASO 5: Entender el funcionamiento

Spring Boot está configurado para servir automáticamente cualquier archivo colocado en:

- static/
- public/
- resources/
- META-INF/resources/

Esto significa que al poner un archivo estático ahí:

- el servidor embebido (Tomcat) lo devuelve tal cual.
- no pasa por ningún controlador.
- no necesita anotaciones.
- no tienes que hacer un @GetMapping.

Los pasos que sigue la ejecución de la aplicación son los siguientes:

- **Inicio de la aplicación:** Se ejecuta el método main, lo que inicia un servidor web embebido (por defecto, Tomcat) en el puerto 8080.
- **Solicitudes HTTP:** En nuestro caso la aplicación solamente está disponible en /hello y cuando un cliente envía una solicitud GET a <http://localhost:8080/hello> (con o sin el parámetro myName), el método sayHello maneja la solicitud. <http://localhost:8080> dará error porque no hay ningún recurso raíz definido.
- **Respuesta:** La aplicación devuelve un mensaje personalizado en texto plano según el parámetro myName .

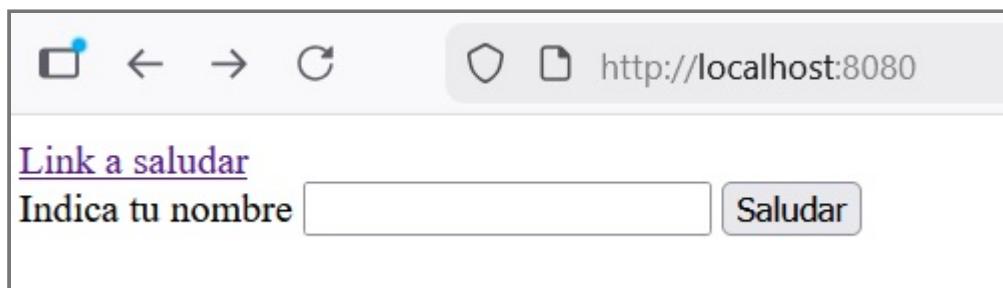
PASO 6: Añadir una página de inicio HTML

Creamos el archivo index.html en src/main/resources/static/ y sustituimos su contenido por:

```
<!DOCTYPE HTML>
<html>
<head>
    <title>Saludo</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
<a href="/hello">Link a saludar</a>

<form action="/hello" method="GET" id="nameForm">
    <div>
        <label for="nameField">Indica tu nombre</label>
        <input name="myName" id="nameField">
        <button>Saludar</button>
    </div>
</form>
</body>
</html>
```

Ahora la aplicación ya se ejecutará en <http://localhost:8080> y servirá index.html como recurso raíz.



Prueba y analiza el ejemplo 1

1. Crea un proyecto Spring Boot llamado `saludo` utilizando Spring Initializr.
2. Prueba el código del ejemplo, verifica que funciona correctamente y pregunta tus dudas.
3. Modifica el archivo `index.html` utilizando css para que tenga una apariencia distinta a la del ejemplo.

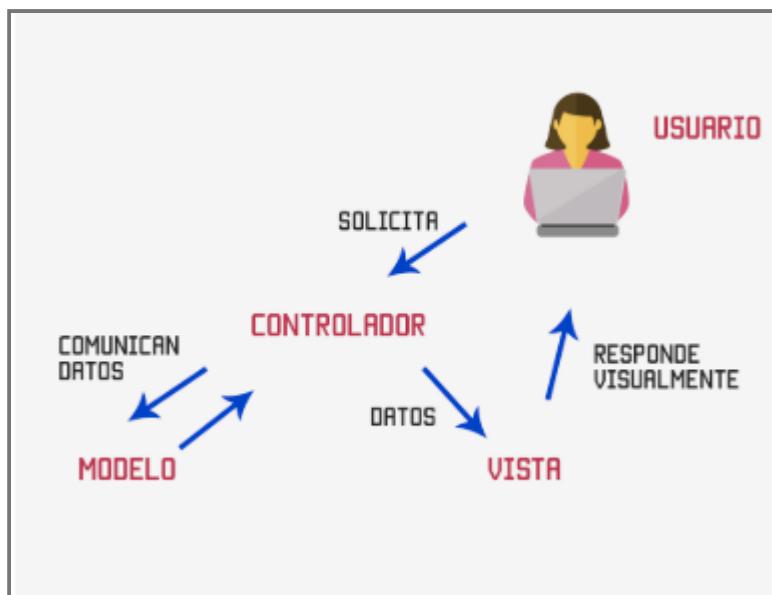
2.3.3 4.3. Spring MVC

Spring MVC es el módulo de Spring orientado al desarrollo de aplicaciones web siguiendo el patrón **Modelo-Vista-Controlador (MVC)**, el cual organiza una aplicación en tres **componentes principales**:

- **Modelo:** Son los datos. Es responsable de:
 - Gestionar el estado de la aplicación.
 - Interactuar con la base de datos u otros servicios para obtener y procesar datos.
 - Proveer datos a la vista.
- **Vista:** Es lo que ve el usuario. Es responsable de:
 - Renderizar información en un formato adecuado, como HTML.
 - Mostrar al usuario los resultados de las acciones ejecutadas.
- **Controlador:** Actúa como intermediario entre el modelo y la vista. Es responsable de:
 - Procesar las solicitudes del usuario (peticiones HTTP).
 - Interactuar con el modelo para obtener o modificar datos.
 - Seleccionar y devolver la vista adecuada para responder al usuario.

Estos tres componentes trabajan de la siguiente forma:

- 1) El usuario interactúa con la **Vista** (interfaz). Envía un formulario o hace clic en un enlace.
- 2) La petición es enviada al **Controlador**.
- 3) El **Controlador** procesa la petición, interactúa con el **Modelo** si es necesario y selecciona la **Vista** que debe renderizar la respuesta.
- 4) La **Vista** presenta la respuesta al usuario.

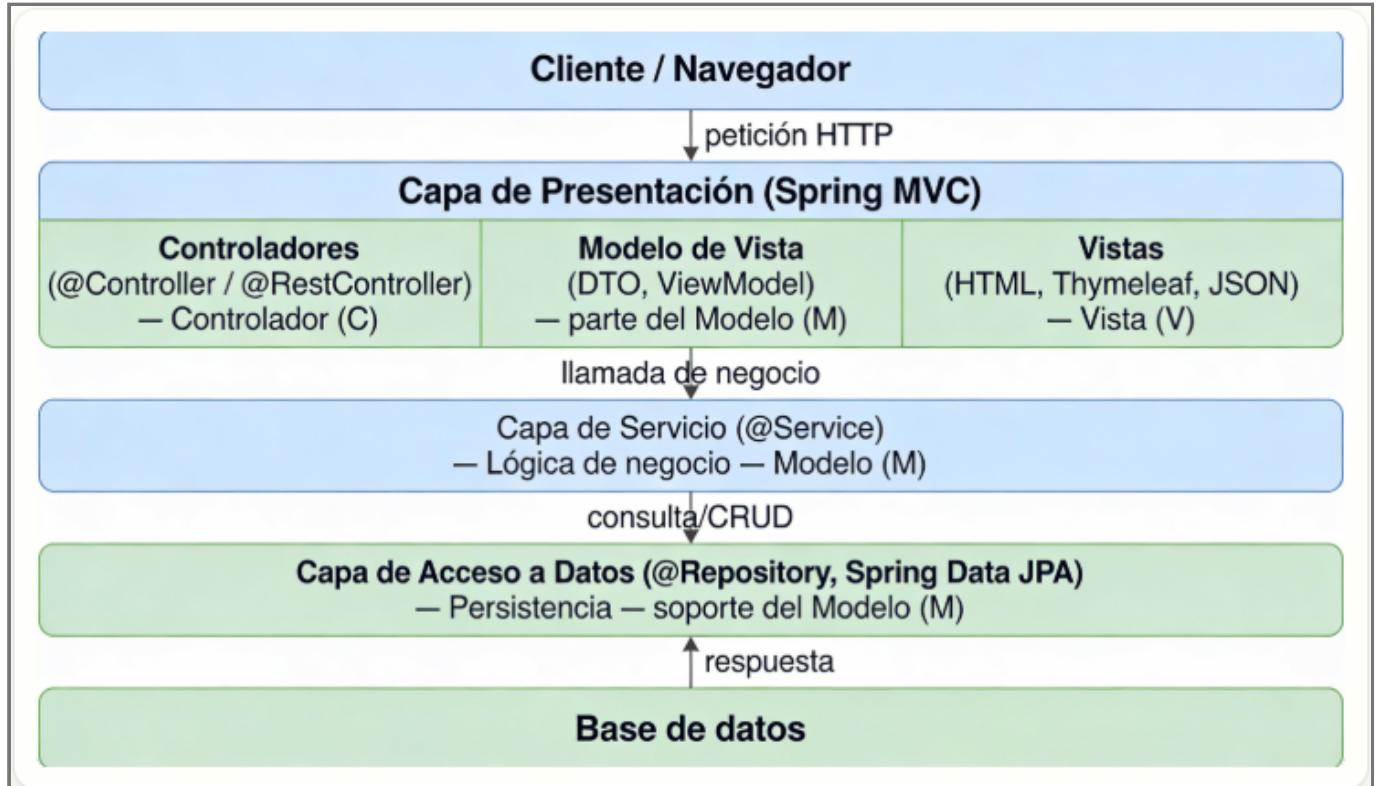


Spring MVC forma parte del ecosistema Spring y se organiza siguiendo una arquitectura en capas en la que cada capa tiene una función concreta y se comunica únicamente con las capas adyacentes. Esta arquitectura encaja perfectamente con el patrón MVC (Model-View-Controller) y proporciona toda la infraestructura necesaria para manejar peticiones HTTP, invocar controladores y devolver vistas (HTML, JSON, etc.) lo que permite aplicaciones más mantenibles, escalables y fáciles de entender.

En la siguiente tabla se muestran las capas más habituales en una aplicación Spring con su equivalencia en Spring MVC, sus anotaciones más habituales y la función que realiza cada una de ellas:

Anotaciones por capa y correspondencia Spring ↔ MVC

Capas Spring	Capa MVC	Anotaciones	Función
Controller (Web)	Controller	@Controller @RestController @RequestMapping @GetMapping @RequestParam @PostMapping @PutMapping @DeleteMapping	Recibe peticiones HTTP, gestiona rutas y parámetros, llama a la capa Service y devuelve una vista o una respuesta (JSON) No contiene lógica de negocio ni acceso a datos
Model (Entidades) Service (Negocio) Repository (Persistencia)	Model	@Entity , @Table , @Id @Service , @Transactional @Repository	Contiene las clases que modelan la información del negocio, aplica reglas y validaciones y accede a la base de datos para realizar operaciones CRUD (manteniendo aislada la BD del resto de la aplicación)
View (Representación HTML / JSON)	View	(sin anotaciones)	Representa los datos al usuario: <ul style="list-style-type: none"> Archivo HTML con sintaxis específica para contenido dinámico si se utiliza Thymeleaf / JSP (Ubicación Thymeleaf: <code>src/main/resources/templates/</code>) Datos en formato JSON / XML en apps REST (si no se utiliza un motor de plantillas). En REST, el JSON actúa como la vista



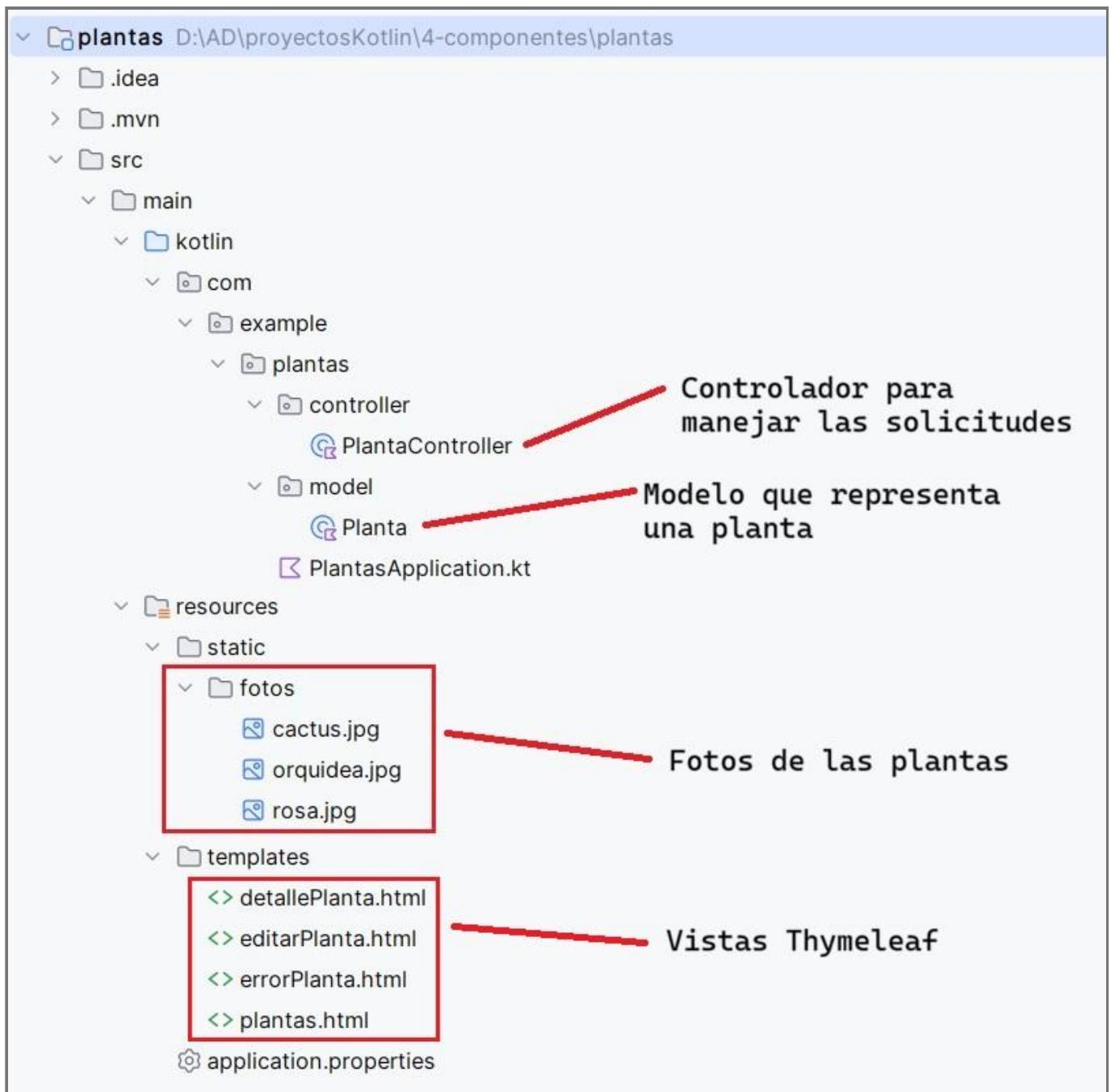
Vistas con Thymeleaf

Thymeleaf es un motor de plantillas que permite mezclar HTML con datos dinámicos proporcionados por el controlador en Spring MVC. Utiliza atributos especiales que comienzan con th: para manipular estos datos de forma dinámica. La siguiente tabla muestra los atributos Thymeleaf más comunes:

Atributo	Descripción	Ejemplo
<code>th:text</code>	Rellena el contenido de un elemento HTML con un valor dinámico.	<code><p th:text="\${mensaje}">Texto por defecto</p></code>
<code>th:each</code>	Itera sobre una colección (lista, array, etc.) y genera un nuevo elemento HTML para cada ítem.	<code><li th:each="planta : \${plantas}" th:text="\${planta.nombre}">Nombre de la planta</code>
<code>th:if</code>	Muestra el contenido solo si la condición es verdadera.	<code><p th:if="\${hayPlantas}">Hay plantas registradas</p></code>
<code>th:unless</code>	Muestra el contenido solo si la condición es falsa.	<code><p th:unless="\${hayPlantas}">No hay plantas registradas</p></code>
<code>th:href</code>	Construye enlaces dinámicos para el atributo href de un enlace <code><a></code> .	<code><a th:href="@{/planta/{id}{id=\${planta.id}}}">Ver detalles</code>
<code>th:src</code>	Construye enlaces dinámicos para el atributo src de una imagen <code></code> .	<code></code>
<code>th:action</code>	Define la URL a la que se enviará un formulario cuando se haga submit.	<code><form th:action="@{/planta/guardar}" method="post"><button type="submit">Guardar</button></form></code>
<code>th:object</code>	Asocia un objeto del modelo con el formulario, permitiendo vincular automáticamente sus atributos.	<code><form th:object="\${planta}" th:action="@{/planta/guardar}" method="post">...</form></code>
<code>th:value</code>	Rellena el valor de un campo de formulario (<code>input</code> , <code>textarea</code> , etc.) con un valor dinámico.	<code><input type="text" th:value="\${planta.nombre}" /></code>
<code>th:field</code>	Asocia un campo de formulario con un atributo del modelo de Spring, vincula los datos automáticamente.	<code><input type="text" th:field="*{nombre}" /></code>

Ejemplo 2: Aplicación utilizando Spring MVC y Thymeleaf

A continuación se describen los pasos para crear una aplicación que muestra una lista con nombres de plantas y junto a cada nombre un enlace que mostrará los detalles de la planta. Desde la pantalla de detalles, se podrá acceder a un formulario para modificar la información de la planta. La estructura del proyecto será la siguiente:



PASO 1: Crear el proyecto

Accedemos a Spring Initializr desde la url <https://start.spring.io/>, indicamos el nombre de la aplicación `plantas` y, en este caso, además de la dependencia **Spring Web** necesitamos también **Thymeleaf** (el resto de opciones las podemos dejar como se ve en la imagen). Por último hacemos clic en el botón GENERATE para descargar nuestro nuevo proyecto.

Opcionalmente podemos añadir **Spring Boot DevTools** que nos ahorrará tiempo de desarrollo ya que:

- Reinicia automáticamente la aplicación cuando cambias código.
- Recarga las plantillas Thymeleaf sin reiniciar manualmente.

Para tener estas funciones activas, además de añadir la dependencia, hay que configurar IntelliJ para que compile al guardar. Esto se consigue activando las opciones siguientes:

- Build project automatically (Settings → Build, Execution, Deployment → Compiler)
- Allow auto-make to start even if developed application is currently running (Settings → Advanced Settings)

De esta forma, cuando realicemos un cambio en un archivo de código de nuestra aplicación, bastará con guardarlo y recargar el navegador (sin reiniciar la app) para ver los cambios inmediatamente.

Project

- Gradle - Groovy
- Gradle - Kotlin
- Maven

Language

- Java
- Kotlin
- Groovy

Dependencies

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Thymeleaf TEMPLATE ENGINES
A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

Spring Boot DevTools DEVELOPER TOOLS
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

Project Metadata

Group	com.example	Artifact	plantas
Name	plantas		
Description	Demo project for Spring Boot		
Package name	com.example.plantas		
Packaging	<input checked="" type="radio"/> Jar	<input type="radio"/> War	
Configuration	<input checked="" type="radio"/> Properties	<input type="radio"/> YAML	
Java	<input type="radio"/> 25	<input checked="" type="radio"/> 21	<input type="radio"/> 17

GENERATE CTRL + ⌘ | **EXPLORE** CTRL + SPACE | ...

PASO 2: Abrir el proyecto

Descomprimimos el archivo obtenido en el paso anterior y abrimos el proyecto con IntelliJ. Comprobamos que la clase principal de la aplicación es `PlantasApplication.kt`, que se encuentra en la carpeta `src/main/kotlin/com/example/plantas/` y que contiene el siguiente código:

```
package com.example.plantas

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class PlantasApplication

fun main(args: Array<String>) {
    runApplication<PlantasApplication>(*args)
}
```

PASO 3: Añadir el controlador

En este caso, como la aplicación es muy sencilla el controlador maneja las solicitudes al recibir las peticiones HTTP, decide qué datos se usan y devuelve la vista adecuada. Para añadir el controlador, creamos el archivo `PlantaController.kt` dentro de la carpeta `src/main/kotlin/com/example/plantas/controller/` con el código siguiente:

```
package com.example.plantas.controller

import com.example.plantas.model.Planta
import org.springframework.stereotype.Controller
import org.springframework.ui.Model
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.PathVariable
import org.springframework.web.bind.annotation.PostMapping

@Controller
class PlantaController {

    private val plantas = mutableListOf(
        Planta(1, "Rosa", "Flor", 0.5, "rosa.jpg"),
        Planta(2, "Cactus", "Succulenta", 1.2, "cactus.jpg"),
        Planta(5, "Orquídea", "Flor", 0.3, "orquidea.jpg")
    )

    @GetMapping("/plantas")
    fun mostrarPlantas(model: Model): String {
        model.addAttribute("plantas", plantas)
        return "plantas" // vista de la lista de plantas (Nombre del archivo HTML en src/main/resources/templates)
    }

    // Detalles de una planta
    @GetMapping("/planta/{id_planta}")
    fun verPlanta(@PathVariable id_planta: Int, model: Model): String {
        // Buscar la planta por id
        val planta = plantas.find { it.id_planta == id_planta }

        // Comprobamos si existe
        if (planta == null) {
            return "errorPlanta" // vista de error sencilla
        }

        model.addAttribute("planta", planta)
        return "detallePlanta" // vista de detalle de una planta
    }

    // formulario para modificar
    @GetMapping("/planta/editar/{id_planta}")
    fun editarPlanta(
        @PathVariable id_planta: Int,
        model: Model
    ): String {
        val planta = plantas.find { it.id_planta == id_planta }
        ?: return "errorPlanta"

        model.addAttribute("planta", planta)
        return "editarPlanta"
    }

    @PostMapping("/planta/guardar")
    fun guardarCambios(plantaModificada: Planta): String {
        val planta = plantas.find { it.id_planta == plantaModificada.id_planta }

        if (planta != null) {
            planta.nombre = plantaModificada.nombre
            planta.tipo = plantaModificada.tipo
            planta.altura = plantaModificada.altura
            planta.foto = plantaModificada.foto
        }
        return "redirect:/planta/${plantaModificada.id_planta}"
    }
}
```

Explicación del código

`@Controller` Indica a Spring que esta clase maneja peticiones web y devuelve vistas HTML.

@GetMapping Muestra páginas HTML

Función	Descripción
@GetMapping("/plantas")	Muestra una lista de todas las plantas en <code>plantas.html</code> .
@GetMapping("/planta/{id_planta}")	Muestra información de detalle de una planta específica en <code>detallePlanta.html</code> . Si no existe, muestra <code>errorPlanta.html</code> .
@GetMapping("/planta/editar/{id_planta}")	Carga la planta en un formulario de edición <code>editarPlanta.html</code> .

@PostMapping Procesa el formulario para editar la información de una planta

Función	Descripción
@PostMapping("/planta/guardar")	Actualiza la planta en memoria y redirige al detalle con <code>redirect:/planta/{id}</code> . Se utiliza <code>redirect</code> para evitar el reenvío de formularios

Model Pasa datos a la vista

@PathVariable Lee datos de la URL

PASO 4: Añadir el modelo

El modelo representa los datos que maneja la aplicación. Para añadir el modelo creamos el archivo `Planta.kt` dentro de la carpeta `src/main/kotlin/com/example/plantas/model/` con el código siguiente:

```
package com.example.plantas.model

data class Planta(
    var id_planta: Int,
    var nombre: String,
    var tipo: String,
    var altura: Double,
    var foto: String
)
```

PASO 5: Añadir las vistas con Thymeleaf

Para nuestra aplicación necesitamos cuatro vistas, una para la lista de plantas, otra para el detalle de una planta, una tercera para avisar en caso de producirse un error y la última para modificar la información de la planta. Por tanto tendremos cuatro archivos `html` todos ellos dentro de la carpeta `src/main/resources/templates/`.

- El archivo que mostrará la lista de plantas será `plantas.html` y su código es el siguiente:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Lista de Plantas</title>
</head>
<body>

<div class="container mt-4">
    <h1 class="mb-4">Plantas</h1>
    <p th:if="${plantas.size() > 0}">Aquí tienes una lista de todas las plantas:</p>
    <p th:unless="${plantas.size() > 0}">No se han encontrado plantas.</p>

    <p th:each="planta : ${plantas}">
        <label th:text="${planta.id_planta}">Id de la planta</label> -
        <label th:text="${planta.nombre}">Nombre de la planta</label> (<label th:text="${planta.altura}">Altura de la planta</label> m)

        <!-- Mostrar enlace a la página de detalles de la planta -->
        <a th:href="@{/planta/{id_planta}(id_planta=${planta.id_planta})}">Ver detalles</a>
    </p>
</div>
</body>
</html>
```

- El archivo que mostrará el detalle de una plantas será `detallePlanta.html` y su código es el siguiente:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Detalles de la Planta</title>
</head>
<body>

<!-- Mostrar foto de la planta --&gt;
&lt;img th:src="@{/fotos/{nombreImagen}}{nombreImagen=${planta.foto}}" alt="Foto de la planta" style="width: 200px;"&gt;

&lt;h1 th:text="${planta.nombre}"&gt;Nombre de la planta&lt;/h1&gt;
&lt;p th:text="Tipo: ' + ${planta.tipo}"&gt;Tipo de planta&lt;/p&gt;
&lt;p th:text="Altura: ' + ${planta.altura} + ' metros!"&gt;Altura de la planta&lt;/p&gt;

&lt;a th:href="@{/planta/editar/{id_planta}}{id_planta=${planta.id_planta}}&gt;Modificar planta&lt;/a&gt;

&lt;p&gt;&lt;a th:href="@{/plantas}"&gt;Volver a la lista de plantas&lt;/a&gt;&lt;/p&gt;

&lt;/body&gt;
&lt;/html&gt;</pre>

```

- El archivo que mostrará el aviso en caso de error será `errorPlanta.html` y su código es el siguiente:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Planta no encontrada</title>
</head>
<body>

<h1>Error</h1>

<p>La planta que estás buscando no existe.</p>

<a th:href="@{/plantas}">Volver a la lista de plantas</a>

</body>
</html>
```

- El archivo que mostrará el formulario para modificar la información de una planta será `editarPlanta.html` y su código es el siguiente:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Editar planta</title>
</head>
<body>

<h1>Modificar planta</h1>

<form th:action="@{/planta/guardar}"
      th:object="${planta}"
      method="post">

    <input type="hidden" th:field="*{id_planta}">

    <p><label>Nombre: </label><input type="text" th:field="*{nombre}"></p>
    <p><label>Tipo: </label><input type="text" th:field="*{tipo}"></p>
    <p><label>Altura: </label><input type="number" step="0.1" th:field="*{altura}"></p>
    <p><label>Foto: </label><input type="text" th:field="*{foto}"></p>

    <button type="submit">Guardar cambios</button>
</form>

<p><a th:href="@{/planta/{id_planta}}{id_planta=${planta.id_planta}}>Volver al detalle</a></p>

</body>
</html>
```

Explicación de las vistas Thymeleaf

Condicionales:

- `th:if` muestra un mensaje si hay plantas registradas.
- `th:unless` muestra un mensaje alternativo si no hay plantas.

Iteración sobre la colección:

- `th:each="planta : ${plantas}"` recorre la lista de plantas (plantas) y crea un bloque de código html (en este caso el que hay dentro de la etiqueta `<p>`) para cada planta.

Mostrar datos dinámicos:

- `th:text="${planta.nombre}"` muestra el nombre de la planta.
- `th:text="Tipo: ' + ${planta.tipo}"` concatena el texto "Tipo: " con el tipo de la planta.
- `th:text="Altura: ' + ${planta.altura} + ' metros"` muestra la altura de la planta en metros.

Enlaces dinámicos:

- `th:href="@{/planta/{id_planta}}(id_planta=${planta.id_planta})"` genera un enlace a la página de detalles de la planta usando el `id_planta` de la planta.

Imágenes dinámicas:

- `th:src="@{/fotos/{nombreImagen}}(nombreImagen=${planta.foto})"` carga foto de la planta.

Formulario:

- `th:action="@{/planta/guardar}"` indica la URL a la que se enviarán los datos del formulario cuando se haga submit.
- `th:object="${planta}"` asocia un objeto del modelo de Spring (Model) con el formulario. En este caso `${planta}` hace referencia a la planta que se pasó al modelo desde el controlador: `model.addAttribute("planta", planta)`. Esto permite usar atributos de planta en los campos del formulario.

PASO 6: Añadir las fotos de las plantas

Para poder mostrar las fotos de nuestras plantas en la vista de detalle, hemos guardado las fotos en una carpeta llamada `fotos` dentro de `src/main/resources/static/`.

PASO 7: Comprobar y ejecutar

Ejecutamos la aplicación usando la clase `PlantasApplication.kt` como clase principal y abrimos la url <http://localhost:8080/plantas> en el navegador. Las siguientes imágenes muestran el funcionamiento de nuestra aplicación:

- Lista de plantas:

The screenshot shows a web browser window with the URL `http://localhost:8080/plantas` in the address bar. The page title is "Plantas". The content area displays a message: "Aquí tienes una lista de todas las plantas:" followed by a list of three items:

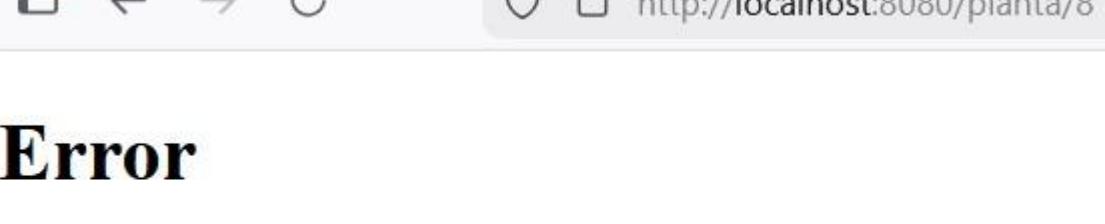
- 1 - Rosa (1.1 m) [Ver detalles](#)
- 2 - Cactus (2.2 m) [Ver detalles](#)
- 5 - Orquídea (3.3 m) [Ver detalles](#)

- Detalle de la planta con `id_planta = 1` (que aparece al hacer clic en el enlace `Ver detalles` junto al nombre de la planta):



A screenshot of a web browser window. The address bar shows the URL <http://localhost:8080/planta/1>. The main content area displays a close-up photograph of a pink rose flower with green leaves in the background. Below the image, the word "Rosa" is displayed in large, bold, black font. Underneath "Rosa", the text "Tipo: Flor" and "Altura: 1.1 metros" are shown. At the bottom, there are two links: "Modificar planta" and "Volver a la lista de plantas".

- Error (en este caso por indicar en la url el id_planta de una planta que no existe):



A screenshot of a web browser window. The address bar shows the URL <http://localhost:8080/planta/8>. The main content area displays the word "Error" in large, bold, black font. Below "Error", the text "La planta que estás buscando no existe." is shown. At the bottom, there is a link "Volver a la lista de plantas".

- Formulario de edición:

The screenshot shows a web browser window with the URL <http://localhost:8080/planta/editar/1>. The page title is "Modificar planta". The form contains the following fields:

- Nombre:
- Tipo:
- Altura: (with up/down arrows)
- Foto:

Below the form are two buttons: "Guardar cambios" and "Volver al detalle".

PASO 8: Cambiar el aspecto

Como hemos visto en las capturas anteriores, nuestras vistas html no tienen aplicado ningún estilo. Vamos a darle a nuestra aplicación un aspecto más profesional utilizando `bootstrap`. Podemos encontrar mucha documentación en internet sobre como utilizarlo. Por ejemplo en:

- <https://getbootstrap.com/docs/5.3/getting-started/introduction/>
- <https://www.w3schools.com/bootstrap5/>

En nuestro caso vamos a descargarlo para incluirlo de forma local en nuestro proyecto y vamos a modificar nuestras vistas `html` para que lo utilicen. Para ello, seguiremos estos pasos:

1. Entrar en <https://getbootstrap.com>
2. Hacer clic en el botón `Download` y descargar la versión **Compiled CSS and JS**
3. Descomprimir el ZIP y copiar la carpeta `bootstrap` en `src/main/resources/static/`
4. Modificar los archivos html para añadir la línea `<link rel="stylesheet" th:href="@{/bootstrap/css/bootstrap.min.css}">` dentro de la etiqueta `<head>` y añadir `<div class="container mt-5">` justo debajo de la etiqueta `<body>` (no olvides añadir también `</div>` justo antes de `</body>`).

Solamente con estos pequeños cambios nuestra aplicación cambiará su aspecto a:

- Lista de plantas:

The screenshot shows a web browser window with the URL `http://localhost:8080/plantas` in the address bar. The page title is "Plantas". The content area displays a message: "Aquí tienes una lista de todas las plantas:" followed by a list of three items:

- 1 - Rosa (0.5 m) [Ver detalles](#)
- 2 - Cactus (1.2 m) [Ver detalles](#)
- 5 - Orquídea (0.3 m) [Ver detalles](#)

- Detalle de la planta con `id_planta = 1` (que aparece al hacer clic en el enlace `ver detalles` junto al nombre de la planta):



A close-up photograph of a single pink rose flower, showing its delicate petals and a few green leaves in the background.

Rosa

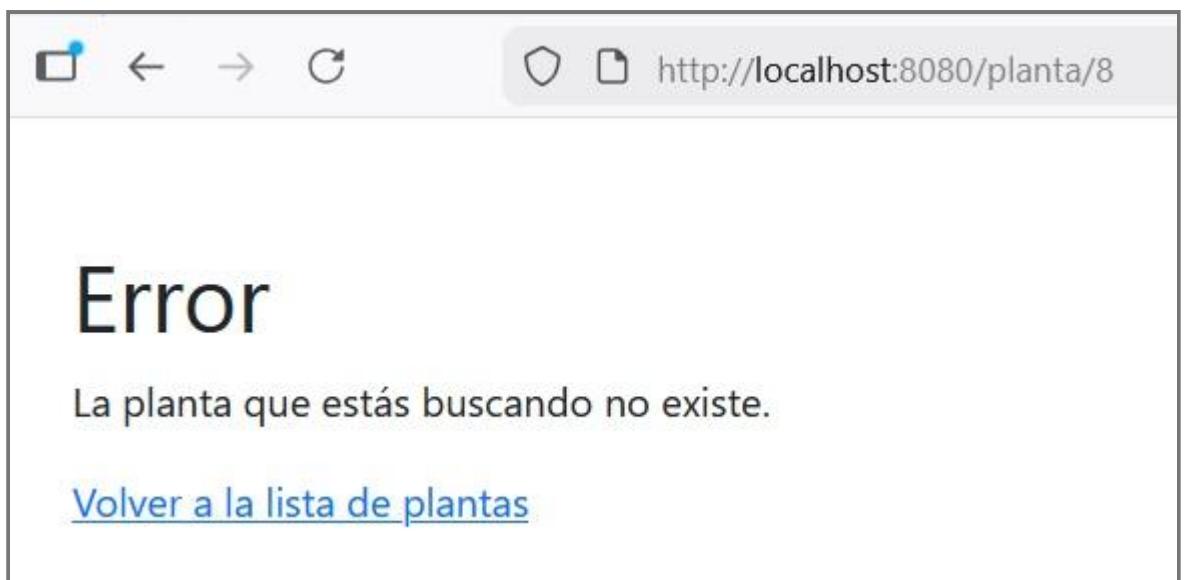
Tipo: Flor

Altura: 0.5 metros

[Modificar planta](#)

[Volver a la lista de plantas](#)

- Error (en este caso por indicar en la url el id_planta de una planta que no existe):



- Formulario de edición:

The screenshot shows a web browser window with the URL <http://localhost:8080/planta/editar/1>. The title of the page is "Modificar planta". The form contains the following fields:

- Nombre:
- Tipo:
- Altura: (with a dropdown arrow icon)
- Foto:
-
- [Volver al detalle](#)



Prueba y analiza el ejemplo 2

1. Crea un proyecto Spring Boot llamado `plantas` utilizando Spring Initializr.
2. Prueba el código del ejemplo, verifica que funciona correctamente y pregunta tus dudas.

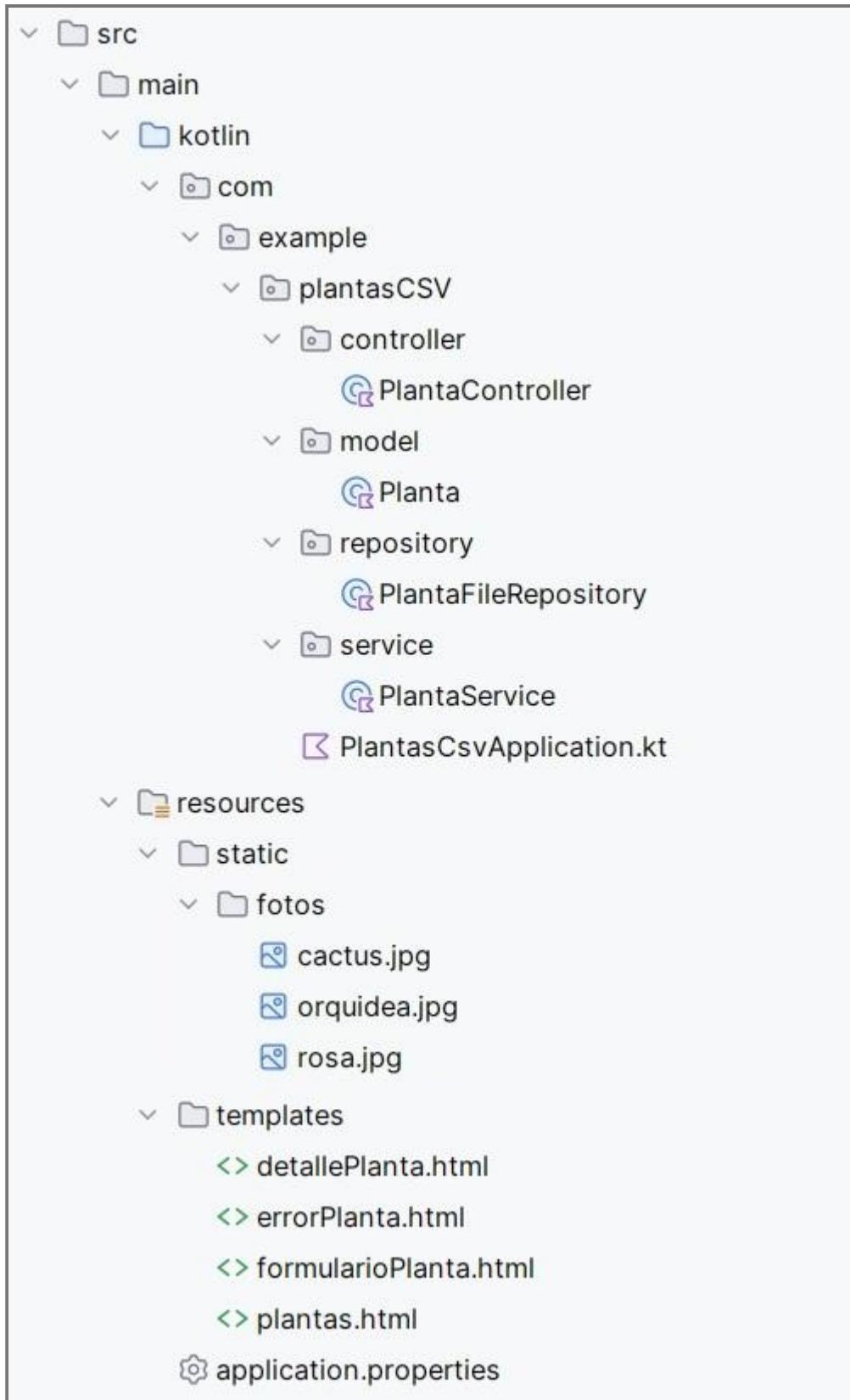
Trabajando con ficheros

En el ejemplo anterior, la información de las plantas se almacenaba en memoria mediante una lista y el controlador accedía directamente a ella. Ahora vamos a trabajar con los datos en un fichero CSV para disponer de persistencia y vamos a separar la responsabilidad de cada capa del patrón MVC de forma que:

- Controlador: interactúa con el usuario.
- Repositorio: maneja los datos.
- Servicio intermedio: hace de intermediario entre el controlador y el repositorio.

Ejemplo 3: CRUD (CSV) con Spring MVC y Thymeleaf

Este ejemplo es un CRUD de información almacenada en un fichero CSV con todas las capas de la arquitectura MVC bien separadas. La estructura del proyecto será la siguiente:



A continuación se describen los pasos necesarios para desarrollar el proyecto:

PASO 1: Crear el proyecto

Creamos un nuevo proyecto llamado `plantascsv` utilizando Spring Initializr con las mismas dependencias del ejemplo anterior. Si copiamos fragmentos de código del anterior ejemplo habrá que tener cuidado con los imports y cambiar `com.example.plantas` por `com.example.plantasCSV`.

PASO 2: Crear el fichero CSV

Creamos un archivo llamado `plantas.csv` con los datos iniciales y lo ubicamos en la carpeta `src/main/resources/data/`. Su contenido inicial será:

```
1;Rosa;Flor;0.5;rosa.jpg
2;Cactus;Suculenta;1.2;actus.jpg
3;Orquidea;Flor;0.3;orquidea.jpg
```

PASO 3: Modificar el controlador

En la arquitectura MVC (Modelo-Vista-Controlador), el controlador es el encargado de recibir las peticiones del usuario (cuando hace clic en un enlace o envía un formulario en el navegador) y decidir qué respuesta dar (normalmente, mostrar una página HTML). Vamos a modificar el controlador que teníamos de la aplicación anterior para que solamente interactúe con el usuario y no acceda a los datos. Además añadiremos el código necesario para las funciones de crear nueva planta y borrar una existente. El código es el siguiente:

```
package com.example.plantasCSV.controller

import com.example.plantasCSV.model.Planta
import com.example.plantasCSV.service.PlantaService

import org.springframework.stereotype.Controller
import org.springframework.ui.Model
import org.springframework.web.bind.annotation.*

@Controller
class PlantaController(
    private val plantaService: PlantaService
) {

    @GetMapping("/plantas")
    fun listar(model: Model): String {
        model.addAttribute("plantas", plantaService.listarPlantas())
        return "plantas"
    }

    @GetMapping("/planta/{id_planta}")
    fun detalle(@PathVariable id_planta: Int, model: Model): String {
        val planta = plantaService.buscarPorId(id_planta)
        ?: return "errorPlanta"

        model.addAttribute("planta", planta)
        return "detallePlanta"
    }

    // CREAR nueva planta
    @GetMapping("/plantas/nueva")
    fun nuevaPlanta(model: Model): String {
        // Pasamos un objeto vacío (o con ID 0/null) para el formulario
        // Como tu data clase tiene tipos primitivos, pon valores por defecto dummy
        val plantaVacia = Planta(0, "", "", 0.0, "")
        model.addAttribute("planta", plantaVacia)
        model.addAttribute("titulo", "Nueva Planta")
        return "formularioPlanta"
    }

    // EDITAR planta existente
    @GetMapping("/plantas/editar/{id_planta}")
    fun editarPlanta(@PathVariable id_planta: Int, model: Model): String {
        val planta = plantaService.buscarPorId(id_planta) ?: return "redirect:/plantas"
        model.addAttribute("planta", planta)
        model.addAttribute("titulo", "Editar Planta")
        return "formularioPlanta"
    }

    // Procesar el GUARDADO (sirve para crear y editar)
    @PostMapping("/plantas/guardar")
    fun guardarPlanta(@ModelAttribute planta: Planta): String {
        plantaService.guardar(planta)
        return "redirect:/plantas"
    }

    // Procesar el BORRADO
    @GetMapping("/plantas/borrar/{id_planta}")
    fun borrarPlanta(@PathVariable id_planta: Int): String {
        plantaService.borrar(id_planta)
        return "redirect:/plantas"
```

```

    }
}
```

PASO 4: Añadir la clase que maneja los datos

Esta clase se encarga de acceder a los datos y gestionarlos, es decir, leer, crear, actualizar y borrar información sobre plantas. Para añadirla creamos el archivo `PlantaFileRepository.kt` dentro de la carpeta `src/main/kotlin/com/example/plantas/repository/` con el siguiente código:

```

package com.example.plantasCSV.repository

import com.example.plantasCSV.model.Planta
import org.springframework.stereotype.Repository
import java.io.File

@Repository
class PlantaFileRepository {

    private val filePath = "src/main/resources/data/plantas.csv"

    fun findAll(): MutableList<Planta> =
        File(filePath).readLines().map { linea ->
            val partes = linea.split(";")
            Planta(
                id_planta = partes[0].toInt(),
                nombre = partes[1],
                tipo = partes[2],
                altura = partes[3].toDouble(),
                foto = partes[4]
            )
        }.toMutableList()

    fun save(planta: Planta) {
        val plantas = findAll()

        // Buscamos si la planta ya existe para saber si es EDITAR o CREAR
        val index = plantas.indexOfFirst { it.id_planta == planta.id_planta }

        if (index != -1) {
            // EDITAR: Reemplazamos la planta existente
            plantas[index] = planta
        } else {
            // CREAR: Calculamos nuevo ID (Simulación de Auto-Increment)
            val nuevoId = (plantas.maxOfOrNull { it.id_planta } ?: 0) + 1
            // Usamos copy porque los val son inmutables, asignando el nuevo ID
            plantas.add(planta.copy(id_planta = nuevoId))
        }
        escribirArchivo(plantas)
    }

    fun deleteById(id_planta: Int) {
        val plantas = findAll()
        plantas.removeIf { it.id_planta == id_planta }
        escribirArchivo(plantas)
    }

    private fun escribirArchivo(plantas: List<Planta>) {
        val contenido = plantas.joinToString(separator = "\n") { planta ->
            "${planta.id_planta};${planta.nombre};${planta.tipo};${planta.altura};${planta.foto}"
        }
        File(filePath).writeText(contenido)
    }
}
```

PASO 5: Añadir la clase del servicio intermedio

En la arquitectura MVC el servicio actúa como el intermediario entre el controlador (parte que interactúa con el usuario) y el repositorio (parte que maneja los datos). Para añadirla creamos el archivo `PlantaService.kt` dentro de la carpeta `src/main/kotlin/com/example/plantas/service/` con el siguiente código:

```

package com.example.plantasCSV.service

import com.example.plantasCSV.model.Planta
import com.example.plantasCSV.repository.PlantaFileRepository
import org.springframework.stereotype.Service

@Service
class PlantaService(
    private val repository: PlantaFileRepository
) {

    fun listarPlantas(): MutableList<Planta> =
        repository.findAll()

    fun buscarPorId(id_planta: Int): Planta? =
        repository.findAll().find { it.id_planta == id_planta }
}
```

```

    fun guardar(planta: Planta) {
        repository.save(planta)
    }

    fun borrar(id_planta: Int) {
        repository.deleteById(id_planta)
    }
}

```

PASO 6: Modificar las vistas

A los archivos html del ejemplo anterior les hemos hecho algunas modificaciones y ahora quedan así:

- El archivo que mostrará la lista de plantas será `plantas.html` y su código es el siguiente:

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Lista de Plantas</title>
    <link rel="stylesheet" th:href="@{/bootstrap/css/bootstrap.min.css}">
</head>

<body>

<div class="container mt-5">
    <h1>Plantas</h1>

    <h5 th:if="${plantas.size() > 0}">Aquí tienes una tabla con todas las plantas y las acciones que puedes hacer con ellas</h5>
    <p th:unless="${plantas.size() > 0}">No hay plantas registradas en el sistema</p>

    <table class="table table-striped">
        <thead>
            <tr>
                <th>ID</th>
                <th>Nombre</th>
                <th>Tipo</th>
                <th>Acciones</th>
            </tr>
        </thead>
        <tbody>
            <tr th:each="planta : ${plantas}">
                <td th:text="${planta.id_planta}">1</td>
                <td th:text="${planta.nombre}">Rosa</td>
                <td th:text="${planta.tipo}">Flor</td>
                <td>
                    <!-- Mostrar enlace a la página de detalles de la planta -->
                    <a th:href="@{/planta/{id_planta}({id_planta=${planta.id_planta}})}" class="btn btn-info btn-sm">Detalles</a>
                    <a th:href="@{/plantas/editar/{id}({id=${planta.id_planta}})}" class="btn btn-info btn-sm">Editar</a>
                    <!-- En un entorno real, borrar debería ser un form con method POST/DELETE,
                        pero para aprender, un GET está bien -->
                    <a th:href="@{/plantas/borrar/{id}({id=${planta.id_planta}})}"
                        class="btn btn-info btn-sm"
                        onclick="return confirm('¿Estás seguro de borrar esta planta?')">Borrar</a>
                </td>
            </tr>
        </tbody>
    </table>
    <a th:href="@{/plantas/nueva}" class="mt-5">Agregar Nueva Planta</a>
</div>
</body>
</html>

```

- El archivo que mostrará el detalle de una plantas será `detallePlanta.html` y su código es el siguiente:

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Detalles de la Planta</title>
    <link rel="stylesheet" th:href="@{/bootstrap/css/bootstrap.min.css}">
</head>

<body>
<div class="container mt-5">
    <div class="row justify-content-center">
        <div class="col-md-6 text-center" style="width: 18rem;">
            <div class="card text-center">
                <!-- Foto -->
                
    <h1 class="card-title text-info mb-4" th:text="${planta.nombre}">Nombre</h1>

    <!-- Datos (usamos fw-bold para negrita) -->
    <p class="fs-5">
        <span class="fw-bold">Tipo:</span>
        <!-- Badge para resaltar el tipo -->
        <span th:text="${planta.tipo}">Tipo</span>
    </p>

    <p class="fs-5">
        <span class="fw-bold">Altura:</span>
        <span th:text="${planta.altura} + ' m'">0.0 m</span>
    </p>

    <hr class="my-4">

    <a th:href="@{/plantas/editar/{id_planta}}(id_planta=${planta.id_planta})" class="btn btn-info bt-sm">Editar</a>

    <a th:href="@{/plantas/borrar/{id_planta}}(id_planta=${planta.id_planta})" class="btn btn-info bt-sm" onclick="return confirm('¿Estás seguro de borrar esta planta?')">Borrar</a>
</div>
</div>
</body>
</html>
```

- El archivo que mostrará el aviso en caso de error será `errorPlanta.html` y su código es el siguiente:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Planta no encontrada</title>
    <link rel="stylesheet" th:href="@{/bootstrap/css/bootstrap.min.css}">
</head>

<body>
<div class="container mt-5">
    <h1>Error</h1>

    <p>La planta que estás buscando no existe.</p>

    <a th:href="@{/plantas}">Volver a la lista de plantas</a>
</div>
```

```
</body>
</html>
```

- El archivo que mostrará el formulario para añadir una planta o modificar la información de una existente será `formularioPlanta.html` y su código es el siguiente:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title th:text="${titulo}">Formulario Planta</title>
    <link rel="stylesheet" th:href="@{/bootstrap/css/bootstrap.min.css}"/>
</head>

<body>
<div class="container mt-5">
<h1 th:text="${titulo}">Formulario</h1>

<form th:action="@{/plantas/guardar}" th:object="${planta}" method="post">

    <!-- Campo oculto para el ID. Si es 0 se creará nuevo, si es &gt; 0 se editarará --&gt;
    &lt;input type="hidden" th:field="*{id_planta}" /&gt;

    &lt;div class="mb-3"&gt;
        &lt;label class="form-label"&gt;Nombre:&lt;/label&gt;
        &lt;input type="text" class="form-control" th:field="*{nombre}" required /&gt;
    &lt;/div&gt;

    &lt;div class="mb-3"&gt;
        &lt;label class="form-label"&gt;Tipo:&lt;/label&gt;
        &lt;input type="text" class="form-control" th:field="*{tipo}" required /&gt;
    &lt;/div&gt;

    &lt;div class="mb-3"&gt;
        &lt;label class="form-label"&gt;Altura (m):&lt;/label&gt;
        &lt;input type="number" step="0.1" class="form-control" th:field="*{altura}" /&gt;
    &lt;/div&gt;

    &lt;div class="mb-3"&gt;
        &lt;label class="form-label"&gt;Nombre Foto (ej: rosa.jpg):&lt;/label&gt;
        &lt;input type="text" class="form-control" th:field="*{foto}" /&gt;
    &lt;/div&gt;

    &lt;button type="submit" class="btn btn-success"&gt;Guardar&lt;/button&gt;
    &lt;a th:href="@{/plantas}" class="btn btn-secondary"&gt;Cancelar&lt;/a&gt;
&lt;/form&gt;
&lt;/div&gt;
&lt;/body&gt;
&lt;/html&gt;</pre>

```

PASO 7: Comprobar y ejecutar

Ejecutamos la aplicación usando la clase `PlantasApplication.kt` como clase principal y abrimos la url <http://localhost:8080/plantas> en el navegador. Las siguientes imágenes muestran el funcionamiento de nuestra aplicación:

- Lista de plantas (en este caso se ha cambiado la lista por una tabla y se han añadido botones de acciones a cada planta):

The screenshot shows a web application interface for managing plants. At the top, there is a header bar with icons for refresh, back, forward, and search, followed by the URL "http://localhost:8080/plantas". To the right of the URL are icons for download, text size, and star.

Plantas

Aquí tienes una tabla con todas las plantas y las acciones que puedes hacer con ellas

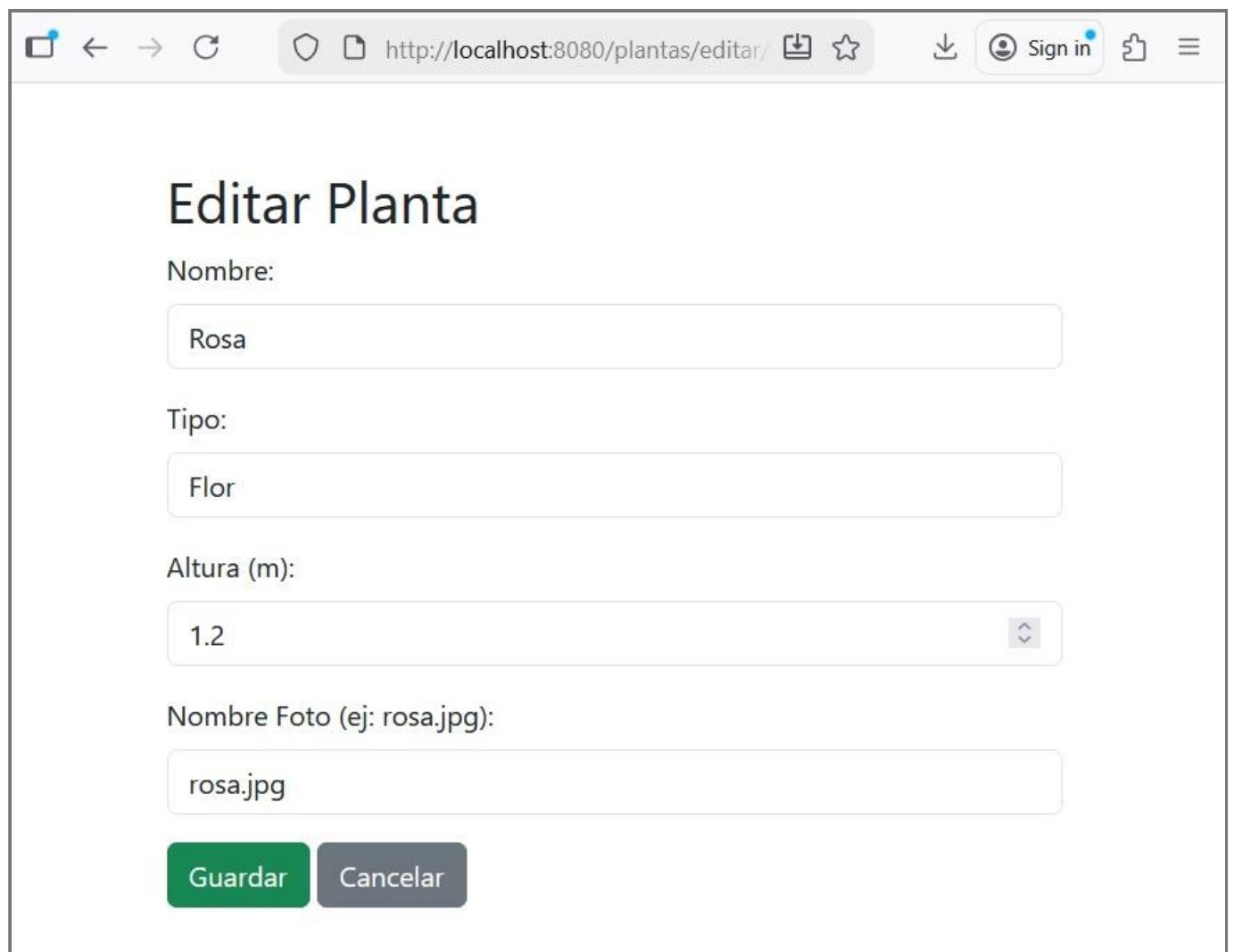
ID	Nombre	Tipo	Acciones
5	Margaritaaaaaa	Flor	Detalles Editar Borrar
6	Rosa	Flor	Detalles Editar Borrar
7	Cáctus	Suculenta	Detalles Editar Borrar
8	Rosa	Flor	Detalles Editar Borrar

[Agregar Nueva Planta](#)

- Detalle de la planta:

The screenshot shows a web browser window with the URL <http://localhost:8080/planta/8>. The page displays a pink rose flower with green leaves. The name of the plant is "Rosa". Below the name, it is categorized as "Tipo: Flor" and its height is listed as "Altura: 1.2 m". At the bottom of the page are two buttons: "Editar" and "Borrar". A link "Volver al listado" is also present.

- Formulario de edición (sirve tanto para añadir una nueva planta como para modificar una ya existente):



Editar Planta

Nombre:

Tipo:

Altura (m):

Nombre Foto (ej: rosa.jpg):

Guardar Cancelar

- Mensaje de confirmación de borrado:

Aquí tienes una tabla con todas las plantas y las acciones que puedes hacer con ellas

ID	Nombre	Tipo	Acciones
5	Margaritaaaaaa	Flor	Detalles Editar Borrar
6	Rosa	Flor	Detalles Editar Borrar
7	Cáctus	localhost:8080	Borrar
8	Rosa		Borrar

[Agregar Nueva Planta](#)

PASO 8: Entender el funcionamiento

Al ejecutar el programa se produce esta secuencia de acciones:

1. El **usuario** entra a /plantas.
2. El **controlador** recibe la petición y llama al **servicio (listarPlantas)**.
3. El **servicio** llama al **repositorio (findAll)**.
4. El **repositorio** lee el archivo plantas.csv, convierte el texto en objetos y los devuelve.
5. El **controlador** mete esos objetos en el **modelo** y carga la plantilla HTML plantas.html.
6. El **usuario** lee la información de las plantas en su navegador.



Prueba y analiza el ejemplo 3

1. Crea un proyecto Spring Boot llamado `plantasCSV` utilizando Spring Initializr.
2. Prueba el código del ejemplo, verifica que funciona correctamente y pregunta tus dudas.



Práctica 1: Trabaja en tu aplicación

1. Crea un nuevo proyecto Spring Boot (con el nombre de tu aplicación) utilizando Spring Initializr.
2. Tu aplicación tendrá las operaciones CRUD sobre un fichero CSV con la información de tu aplicación.
3. Modifica el aspecto de las vistas con `bootstrap` o `css` propio para que el resultado quede totalmente diferente al del ejemplo.

2.3.4 4.4. Spring Data

Spring Data proporciona herramientas y abstracciones para facilitar el acceso a bases de datos y otras fuentes de datos de manera **eficiente y consistente**. Su objetivo principal es **simplificar la interacción con diferentes tipos de bases de datos**,

tanto **relacionales** (como PostgreSQL o MySQL) como **NoSQL** (por ejemplo MongoDB), reduciendo la cantidad de código necesario y **unificando la forma de trabajar con los datos**.

Se utiliza para:

1. Acceso Simplificado a Datos:

- Reduce la necesidad de escribir consultas SQL complejas o código JDBC al exponer métodos predefinidos para operaciones comunes.
- Permite realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) con facilidad.

2. Abstracción de Repositorios: Ofrece la interfaz **Repository** y subinterfaces como **CrudRepository** y **JpaRepository** que proporcionan métodos estándar para la gestión de entidades en bases de datos relacionales.

3. Consultas Personalizadas: Permite escribir consultas personalizadas mediante anotaciones como **@Query**. También admite la creación de métodos de consulta basados en el nombre del método, como **findByNombre(String nombre)**.

4. Compatibilidad con Múltiples Tecnologías de Bases de Datos:

- Relacionales: Mediante JPA (Java Persistence API).
- NoSQL: MongoDB, Redis, Neo4j, Cassandra, etc.
- Buscadores: Elasticsearch, Solr.

5. Configuración Declarativa: Al integrar Spring Data con Spring Boot, se pueden configurar muchas opciones mediante propiedades en **application.properties**, evitando configuraciones manuales detalladas.

6. Integración con Spring Boot: Con dependencias específicas como **spring-boot-starter-data-jpa** o **spring-boot-starter-data-mongodb**, Spring Data se integra perfectamente con el resto del ecosistema de Spring.

En lugar de proporcionar una única solución, Spring Data está compuesto por varios módulos, cada uno diseñado para un tipo concreto de tecnología de persistencia, como bases de datos relacionales, NoSQL o sistemas de búsqueda. Gracias a esta estructura modular, el desarrollador puede cambiar la tecnología de persistencia sin modificar la arquitectura general de la aplicación. Sus principales módulos son:

- **Spring Data JPA:** Proporciona una integración con JPA para bases de datos relacionales. Es ideal para trabajar con entidades Java mapeadas a tablas de bases de datos. JPA es la especificación para persistir, leer y gestionar data desde los objetos Java a la base de datos.
- **Spring Data MongoDB:** Facilita el acceso a bases de datos MongoDB, una base de datos NoSQL orientada a documentos.
- **Spring Data Redis:** Para aplicaciones que necesitan interactuar con Redis, una base de datos en memoria.
- **Spring Data Cassandra:** Proporciona soporte para bases de datos distribuidas como Cassandra.
- **Spring Data Elasticsearch:** Simplifica las interacciones con Elasticsearch, un motor de búsqueda y análisis.

4.4.1. Spring Data JPA (BD relacionales)

Spring Data JPA (Java Persistence API) es un módulo de Spring Data que sirve para simplificar el acceso a bases de datos relacionales utilizando objetos (clases) sin tener que escribir SQL ni código repetitivo. Con Spring Data JPA:

- Solo defines entidades (@Entity)
- Creas interfaces Repository
- Spring genera automáticamente el código

Anotaciones JPA

Algunas de las anotaciones JPA son las siguientes:

Anotaciones de Mapeo JPA

- `@Entity` - Marca una clase como una entidad JPA, mapeada a una tabla en la base de datos.

```
@Entity
data class Planta(
    @Id
    val id: Int,
    val nombre: String,
    val tipo: String,
    val altura: Double
)
```

- `@Table` - Especifica el nombre de la tabla que corresponde a la entidad.

```
@Entity
@Table(name = "plantas")
data class Planta(
    @Id
    var id_planta: Int,
    val nombre: String,
    val tipo: String,
    val altura: Double
)
```

- `@Id` - Indica que un campo es la clave primaria de la tabla.

```
@Id
val id_planta: Int
```

- `@GeneratedValue` - Define cómo se genera el valor de la clave primaria (`.GenerationType.IDENTITY`, `.GenerationType.SEQUENCE`, etc.)

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
val id_planta: Int
```

- `@Column` - Configura una columna de la tabla, como nombre, si es nula o única.

```
@Column(nombre = "planta_nombre", nullable = false, unique = true)
val name: String
```

- `@JoinColumn` - Especifica la columna que actúa como clave foránea.

```
@ManyToOne
@JoinColumn(name = "id_jardin")
var jardin: Jardin
```

- `@Lob` - Indica que un campo es un objeto de gran tamaño (texto o binario).

```
@Lob
val description: String
```

- `@Transient` - Excluye un campo del mapeo de base de datos (no se almacena).

```
@Transient
val calculatedField: String
```

Anotaciones de relaciones JPA

- `@ManyToOne`, `@OneToMany`, `@OneToOne`, `@ManyToMany`: Definen relaciones entre entidades.

```
@ManyToOne
val department: Department
```

Anotación	Descripción
@ManyToOne	Relación muchos a uno
@OneToMany	Relación uno a muchos
@OneToOne	Relación uno a uno
@ManyToMany	Relación muchos a muchos

Anotaciones de Spring Data JPA

- `@Repository` - Marca una interfaz o clase como repositorio Spring. El `_data class_` representa la entidad (tabla), mientras que el `@Repository` se encarga de las operaciones de acceso a datos sobre esa entidad.

```
@Repository
interface PlantaRepository : JpaRepository<Planta, Int>
```

- `@Query` y `@Param` - El primero define una consulta personalizada usando JPQL o SQL nativo y el segundo define parámetros nombrados para consultas con `@Query`.

```
//Ejemplo (JPQL):
@Query("SELECT p FROM Planta p WHERE p.nombre = :nombre")
fun findByNombre(@Param("nombre") nombre: String): List<Planta>

//Ejemplo (SQL nativo):
@Query(value = "SELECT * FROM plantas WHERE nombre = :nombre", nativeQuery = true)
fun findByNombreNative(@Param("nombre") nombre: String): List<Planta>
```

- `@Modifying` - Se utiliza con consultas `@Query` para operaciones de actualización o eliminación.

```
@Modifying
@Query("UPDATE Planta p SET p.nombre = :nombre WHERE p.id_planta = :id_planta")
fun actualizarNombre(
    @Param("id_planta") id_planta: Int,
    @Param("nombre") nombre: String
)
```

- `@EnableJpaRepositories` - Habilita la funcionalidad de Spring Data JPA y escanea paquetes para detectar repositorios.

```
@EnableJpaRepositories(basePackages = ["com.example.repository"])
```

- `@EntityGraph` - Especifica cómo cargar las relaciones en una consulta, evitando lazy loading.

```
//Busca una planta por su nombre y carga la planta + el jardín asociado
@EntityGraph(attributePaths = ["jardin"])
fun findByNombre(nombre: String): Planta?
```

Anotaciones de Transacciones

- `@Transactional` - Marca un método o clase para ejecutar dentro de una transacción.

```
@Transactional
fun updatePlantaDetails(planta: Planta) { ... }
```

- `@Rollback` - Utilizada para forzar la reversión de una transacción.

```
@Transactional
@Rollback
fun testSavePlanta() { ... }
```

Consultas utilizando convención de nombres

Spring Data JPA permite definir métodos en repositorios siguiendo una convención de nombres específica. Esto simplifica la escritura de consultas comunes sin necesidad de usar JPQL o SQL. Para ello analiza el nombre de los métodos en el repositorio e interpreta su significado para generar consultas automáticamente.

La **convención de nombres** se utiliza:

- En consultas sencillas y que no requieren lógica compleja ni múltiples combinaciones.
- Cuando quieras mantener un código más limpio y directo.

La estructura básica es: `findBy + NombreDeCampo + Condición` donde:

- **findBy**: Indica que se busca una entidad en la base de datos. Alternativas: **readBy** (lectura de datos), **queryBy** (consulta de datos) y **getBy** (obtener datos).
- **NombreDeCampo**: Debe coincidir exactamente con el nombre del atributo en la entidad. Se puede incluir navegación de atributos para relaciones (EntidadRelacionada.Atributo).
- **Condición** (opcional): Permite añadir operadores lógicos como And, Or, etc. Ejemplo: `findByNombreAndAltura`.

Ejemplos de métodos según la convención

- Consultas simples: Método **findByNombre(String nombre)**. Ejemplo de consulta generada:

```
SELECT * FROM plantas WHERE nombre = ?
```

- Consultas con condiciones: Método **findByNombreAndAltura(String nombre, Double altura)**. Ejemplo de consulta generada:

```
SELECT * FROM plantas WHERE nombre = ? AND altura = ?
```

- Consultas con orden: Método **findByNombreOrderByAlturaDesc(String nombre)**. Ejemplo de consulta generada:

```
SELECT * FROM plantas WHERE nombre = ? ORDER BY altura DESC
```

- Consultas con relaciones. Si hay una relación entre entidades, se puede navegar por los campos relacionados: Método **findByComarcaNomC(String nomC)**. Ejemplo de consulta generada:

```
SELECT * FROM plantas JOIN jardines ON plantas.id_jardin = jardines.id_jardin WHERE plantas.nombre = ?
```

Palabras clave en la convención

Palabra Clave	Función	Ejemplo
And	Combina múltiples condiciones con AND	findByNombreAndAltura
Or	Combina múltiples condiciones con OR	findByNombreOrAltura
Between	Busca valores en un rango	findByAlturaBetween
LessThan	Busca valores menores a un límite	findByAlturaLessThan
GreaterThan	Busca valores mayores a un límite	findByAlturaGreaterThan
IsNull	Busca valores NULL	findByTipoIsNull
IsNotNull	Busca valores que no sean NULL	findByTipoIsNotNull
Like	Busca valores que coincidan con un patrón	findByNombreLike
NotLike	Busca valores que no coincidan con un patrón	findByNombreNotLike
StartingWith	Busca valores que comiencen con un prefijo	findByNombreStartingWith
EndingWith	Busca valores que terminen con un sufijo	findByNombreEndingWith
Containing	Busca valores que contengan un patrón	findByNombreContaining
OrderBy	Ordena los resultados por un campo específico	findByNombreOrderByAlturaDesc
Top	Devuelve los primeros resultados (límite)	findTop3ByAlturaGreaterThan
First	Devuelve el primer resultado	findFirstByNombre

A tener en cuenta:

- **Coincidencia exacta del nombre del campo:** Los nombres deben coincidir con los atributos definidos en la entidad.
- **Relaciones:** Usa la notación *EntidadRelacionada.Atributo* para navegar entre tablas relacionadas.
- **Orden:** Los métodos pueden incluir palabras clave de ordenación, como *OrderBy*.
- **Parámetros:** Los métodos generados reciben parámetros en el mismo orden en que se declaran en el nombre del método.

Ejemplo 4: CRUD con JPA y MySQL

En este proyecto vamos a almacenar la información de nuestras plantas en una BD MySQL y utilizar JPA para las operaciones CRUD. Los pasos para desarrollar la aplicación son los siguientes:

PASO 1: Crear el proyecto

Creamos un nuevo proyecto llamado `plantasMySQL` utilizando Spring Initializr. En este caso añadimos la dependencia JPA como se ve en la imagen siguiente:

Project

- Gradle - Groovy
- Maven
- Gradle - Kotlin
- Groovy

Language

- Java
- Kotlin
- Groovy

Spring Boot

- 4.0.2 (SNAPSHOT)
- 4.0.1
- 3.5.10 (SNAPSHOT)
- 3.5.9

Project Metadata

Group	com.example	Artifact	plantasMySQL
Name	plantasMySQL		
Description	Demo project for Spring Boot		
Package name	com.example.plantasMySQL		
Packaging	<input checked="" type="radio"/> Jar	<input type="radio"/> War	
Configuration	<input checked="" type="radio"/> Properties	<input type="radio"/> YAML	
Java	<input type="radio"/> 25	<input checked="" type="radio"/> 21	<input type="radio"/> 17

Dependencies

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Thymeleaf TEMPLATE ENGINES
A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

Spring Boot DevTools DEVELOPER TOOLS
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

Spring Data JPA SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

ADD DEPENDENCIES... CTRL + B

GENERATE CTRL + ↵ **EXPLORE** CTRL + SPACE **...**

Hay que tener cuidado si copiamos fragmentos de código del anterior ejemplo ya que habrá que cambiar en los imports `com.example.plantasCSV` por `com.example.plantasMySQL`.

PASO 2: Añadir dependencias y configuración

Añadimos las siguientes líneas al archivo `pom.xml` dentro del bloque `<dependencies>`:

```
<!-- MySQL Connector -->
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <scope>runtime</scope>
</dependency>
```

Añadimos las siguientes líneas al archivo `application.properties` sustituyendo `BD` por el nombre de nuestra base de datos, `user` por el nombre de usuario y `pass` por la contraseña (los que creamos al montar el servidor y crear la base de datos):

```
# =====
# DATASOURCE (MySQL)
# =====
spring.datasource.url=jdbc:mysql://localhost:3306/nom_BD?useSSL=false&allowPublicKeyRetrieval=true
spring.datasource.username=user
spring.datasource.password=pass
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# =====
# JPA / HIBERNATE
# =====
spring.jpa.database-platform=org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

PASO 3: Añadir el modelo

Ahora nuestra clase Planta debe ser una entidad JPA. En Kotlin, JPA necesita un constructor vacío, por tanto, debemos asignar valores por defecto a todos los campos de la data class para que Kotlin genere ese constructor automáticamente.

El contenido del archivo `src/main/kotlin/com/example/plantasMySQL/model/Planta.kt` es el siguiente:

```
package com.example.plantasMySQL.model
import jakarta.persistence.*

@Entity
@Table(name = "plantas")
data class Planta(
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    var id_planta: Int? = null,
    @Column(nullable = false)
    var nombre: String = "",
    @Column
    var tipo: String = "",
    @Column
    var altura: Double = 0.0,
    @Column
    var foto: String = ""
)
```

PASO 4: Añadir el repositorio

El contenido del archivo `src/main/kotlin/com/example/plantasMySQL/repository/PlantaRepository.kt` es el siguiente:

```
package com.example.plantasMySQL.repository

import com.example.plantasMySQL.model.Planta
import org.springframework.data.jpa.repository.JpaRepository
import org.springframework.stereotype.Repository

@Repository
interface PlantaRepository : JpaRepository<Planta, Int>
```

PASO 5: Añadir el servicio

El código del servicio se simplifica enormemente (ahora llamará a los métodos de PlantaRepository). El contenido del archivo `src/main/kotlin/com/example/plantasMySQL/service/PlantaService.kt` es el siguiente:

```
package com.example.plantasMySQL.service

import com.example.plantasMySQL.model.Planta
import com.example.plantasMySQL.repository.PlantaRepository
import org.springframework.stereotype.Service
import kotlin.compareTo

import java.io.File
import kotlin.collections.map
import kotlin.io.readLines
import kotlin.text.split
import kotlin.text.toDouble

@Service
class PlantaService(private val repository: PlantaRepository) {

    fun listarPlantas(): List<Planta> = repository.findAll()

    fun buscarPorId(id: Int): Planta? = repository.findById(id).orElse(null)

    fun guardar(planta: Planta): Planta = repository.save(planta)

    fun borrar(id: Int) {
        if (repository.existsById(id)) {
            repository.deleteById(id)
        }
    }

    fun importarDesdeCSV() {
        // 1. Verificar si ya hay datos para no duplicar (opcional)
        //if (repository.count() > 0) return

        // 2. Leer el fichero (Lógica del Ejemplo 3)
        val filePath = "src/main/resources/data/plantas.csv"
        val file = File(filePath)

        if (file.exists()) {
```

```

        println("el fichero existe")
        val plantasLeidas = file.readLines().map { linea ->
            val partes = linea.split(";")
            // 3. Crear el objeto Planta (Lógica del Ejemplo 4)
            // IMPORTANTE: Ponemos id_planta = null para que se genere uno nuevo
            Planta(
                id_planta = null,
                nombre = partes[1],
                tipo = partes[2],
                altura = partes[3].toDouble(),
                foto = partes[4]
            )
        }
        // 4. Guardar todo en la BD de golpe
        repository.saveAll(plantasLeidas)
    }
}
}

```

PASO 6: Añadir el controlador

El contenido del archivo `src/main/kotlin/com/example/plantasMySQL/controller/PlantaController.kt` es el siguiente:

```

package com.example.plantasMySQL.controller
import com.example.plantasMySQL.model.Planta
import com.example.plantasMySQL.service.PlantaService
import org.springframework.stereotype.Controller
import org.springframework.ui.Model
import org.springframework.web.bind.annotation.*

@Controller
class PlantaController(private val plantaService: PlantaService) {

    @GetMapping("/plantas")
    fun listar(model: Model): String {
        model.addAttribute("plantas", plantaService.listarPlantas())
        return "plantas"
    }

    @GetMapping("/planta/{id_planta}")
    fun detalle(@PathVariable id_planta: Int, model: Model): String {
        val planta = plantaService.buscarPorId(id_planta) ?: return "errorPlanta"
        model.addAttribute("planta", planta)
        return "detallePlanta"
    }

    @GetMapping("/plantas/nueva")
    fun nuevaPlanta(model: Model): String {
        val plantaVacia = Planta(nombre = "", tipo = "", altura = 0.0, foto = "")
        model.addAttribute("planta", plantaVacia)
        model.addAttribute("titulo", "Nueva Planta")
        return "formularioPlanta"
    }

    @GetMapping("/plantas/editar/{id_planta}")
    fun editarPlanta(@PathVariable id_planta: Int, model: Model): String {
        val planta = plantaService.buscarPorId(id_planta) ?: return "redirect:/plantas"
        model.addAttribute("planta", planta)
        model.addAttribute("titulo", "Editar Planta")
        return "formularioPlanta"
    }

    @PostMapping("/plantas/guardar")
    fun guardarPlanta(@ModelAttribute planta: Planta): String {
        plantaService.guardar(planta)
        return "redirect:/plantas"
    }

    @GetMapping("/plantas/borrar/{id_planta}")
    fun borrarPlanta(@PathVariable id_planta: Int): String {
        plantaService.borrar(id_planta)
        return "redirect:/plantas"
    }

    /*
     * 1. Ruta para mostrar la página de inicio (index.html)
     */
    @GetMapping("/")
    fun inicio(): String {
        return "index"
    }

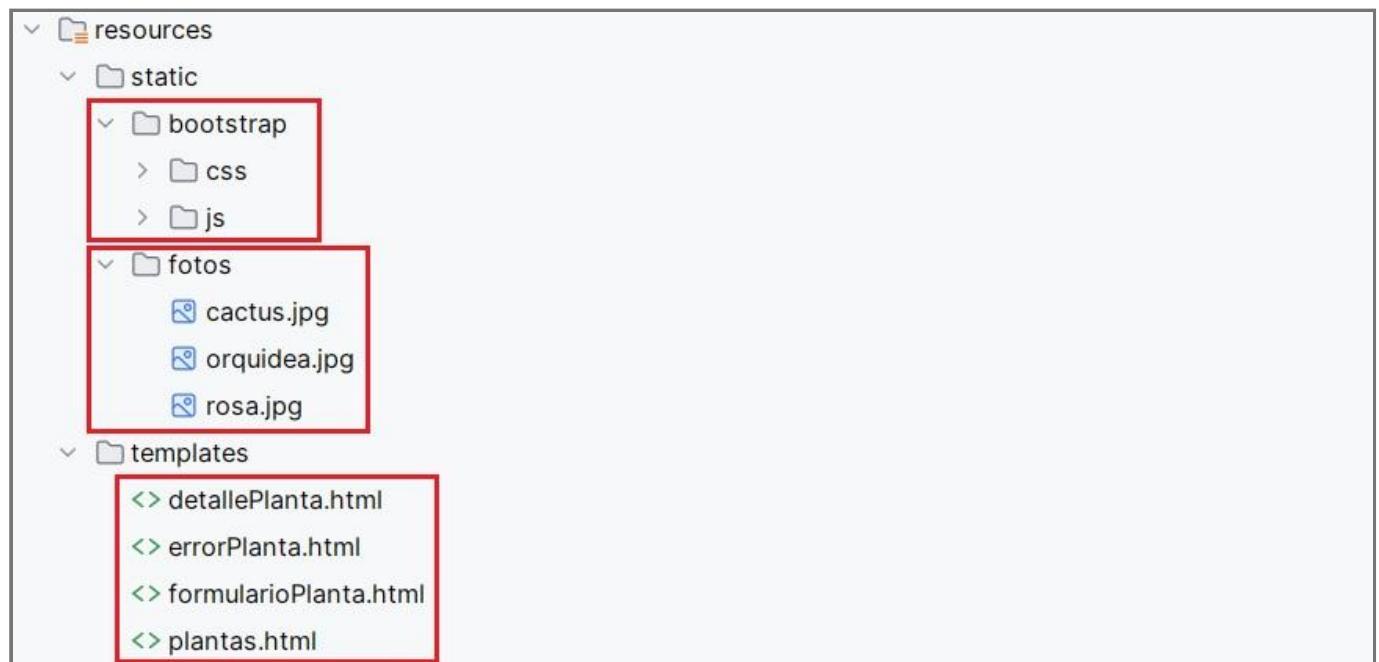
    /*
     * 2. Ruta para ejecutar la importación
     */
    @GetMapping("/importar")
    fun importarDatos(): String {
        plantaService.importarDesdeCSV() // <-- Tienes que crear este método en el Service como hablamos antes
        return "redirect:/plantas" // Al terminar, nos lleva a la lista para ver el resultado
    }
}

```

}

PASO 7: Añadir las vistas, bootstrap y las imágenes

Todos estos archivos los podemos copiar del proyecto del ejemplo anterior, son los siguientes:

**PASO 8: Comprobar y ejecutar**

Al ejecutar la aplicación `PlantaApplication.kt`, por primera vez veremos en la consola mensajes de Hibernate: `Hibernate: create table plantas (...)` que indican que se ha creado la tabla `plantas` dentro de la base de datos llamada `florabotanica.db`.

Al abrir <http://localhost:8080/plantas> en el navegador veremos lo siguiente (inicialmente la tabla plantas estará vacía):

ID	Nombre	Tipo	Acciones
Agregar Nueva Planta			

Podemos añadir tantas plantas como queramos con el botón `Agregar Nueva Planta` y comprobar que los datos se guardan en la BD y son persistentes si detenemos la aplicación y la volvemos a arrancar.

✓ Prueba y analiza el ejemplo 4

1. Monta un servidor MySQL en Docker.
2. Crea un proyecto Spring Boot llamado `plantasMySQL` utilizando Spring Initializr.
3. Prueba el código del ejemplo, verifica que funciona correctamente y pregunta tus dudas.

⚠️ Práctica 2: Trabaja en tu aplicación

1. Crea un nuevo proyecto Spring Boot (con el nombre de tu aplicación) utilizando Spring Initializr.
2. Tu aplicación tendrá una vista principal (`index.html`) con las opciones siguientes:
 - Importar información: Leerá información de un fichero CSV y la guardará en una tabla de una BD MySQL.
 - CRUD: funcionará como el ejemplo 4 con las operaciones CRUD sobre la BD MySQL.
3. Modifica el aspecto de tu aplicación aplicando alguna característica de `bootstrap` para que el resultado quede personalizado a tu gusto.

Consultas utilizando la anotación `@Query`

La anotación `@Query` se utiliza:

- En consultas complejas que involucren múltiples tablas, condiciones avanzadas o subconsultas.
- Si prefieres optimizar manualmente las consultas.
- Cuando la convención de nombres generaría un nombre de método excesivamente largo.

La estructura básica es:

```
@Query("SELECT e FROM EntityName e WHERE e.property = :value")
fun findByProperty(@Param("value") value: String): List<EntityName>
```

Hay que tener en cuenta que se utiliza JPQL y no SQL

JPQL	SQL
Pregunta	al modelo de objetos
trabaja	con clases y atributos
utiliza	nombres de entidades y propiedades de las clases
	nombres de tablas y columnas

JPQL

- Puede navegar por relaciones entre entidades.
- Utiliza `:nombreParametro` para parámetros dinámicos.

Ejemplo sencillo: Devuelve todas las plantas cuyo tipo coincide con el valor indicado.

```
@Query("SELECT p FROM Planta p WHERE p.tipo = :tipo")
fun findByTipo(@Param("tipo") tipo: String): List<Planta>
```

Ejemplo con relaciones:

La consulta para buscar las plantas que están asociadas con un jardín específico (según el ID del jardín proporcionado):

```
@Query("""
    SELECT jp
    FROM JardinPlanta jp
    JOIN jp.planta p
    WHERE jp.jardin.id_jardin = :idJardin
    ORDER BY p.nombre
""")
```

```
""")  
fun obtenerPlantasDeJardin(  
    @Param("idJardin") idJardin: Int  
) : List<JardinPlanta>
```

SELECT: ¿qué se devuelve?

`SELECT jp` (objetos completos y no campos)

FROM: ¿de dónde salen los datos?

`FROM JardinPlanta jp` (de la clase `JardinPlanta`)

En SQL sería: `FROM jardines_plantas jp` (de la tabla `jardines_plantas`)

JOIN: ¿cómo se relacionan los datos?

`JOIN jp.planta p` (se usa la relación que ya existe entre los objetos)

En SQL sería: `JOIN plantas p ON jp.id_planta = p.id_planta` (se relacionan dos tablas usando claves foráneas)

WHERE: ¿qué se filtra?

`WHERE jp.jardin.id_jardin = :idJardin` (se navega por objetos de la relación → al objeto → a su atributo)

En SQL sería: `WHERE jp.id_jardin = :idJardin` (se filtra por una columna)

ORDER BY: ¿cómo se ordena?

`ORDER BY p.nombre` (se ordena por un atributo de la entidad)

En SQL sería: `ORDER BY p.nombre` (por una columna)

Este mismo ejemplo utilizando convención de nombres quedaría así:

```
@Repository  
interface PlantaRepository : JpaRepository<Planta, Int> {  
    fun findByJardinIdJardin(  
        idJardin: Int  
    ): List<Planta>  
}
```

- **findBy**: Indica que es un método de consulta.

- **findByJardinIdJardin**: Utiliza la convención de nombres para especificar que se desean encontrar plantas basándose en el `idJardin` del jardín asociado. Los nombres de los parámetros reflejan claramente los campos utilizados en la consulta.

Ejemplo 5: Consultas avanzadas con @Query (MySQL)

En el ejemplo anterior hemos utilizado **Spring Data JPA** junto con **MySQL** para implementar un CRUD completo apoyándonos en los métodos que ofrece la interfaz `JpaRepository` y en **consultas generadas automáticamente mediante convención de nombres**.

Este mecanismo es suficiente para consultas sencillas, pero cuando las consultas incluyen **múltiples condiciones, ordenaciones, actualizaciones directas** o requieren el uso de **SQL nativo**, resulta más adecuado utilizar la anotación `@Query`.

En esta ampliación se utiliza `@Query` para realizar consultas avanzadas con SQL nativo sobre nuestra base de datos. Las operaciones `UPDATE` y `DELETE` requieren el uso de `@Modifying` y `@Transactional`.

Para poder ilustrar la utilización de este tipo de consultas, ampliaremos nuestra aplicación para gestionar las plantas que hay en los jardines. Imagina que a nuestra empresa de jardinería la contratan varios ayuntamientos que nos piden trabajos del tipo "hay que plantar 2 rosas y 3 pinos en el jardín llamado Tropical" y "hay que plantar 5 rosas y 4 pinos en el jardín llamado Botánico". Para ello ampliaremos con dos tablas más nuestra base de datos y con varios archivos nuestra aplicación. A continuación describen todos los pasos necesarios:

PASO 1: Crear el proyecto

Podemos crear un proyecto nuevo con las mismas dependencias que el del ejemplo anterior y copiar los archivos que tenemos o ampliar el ejemplo anterior directamente.

PASO 2: Añadir los jardines

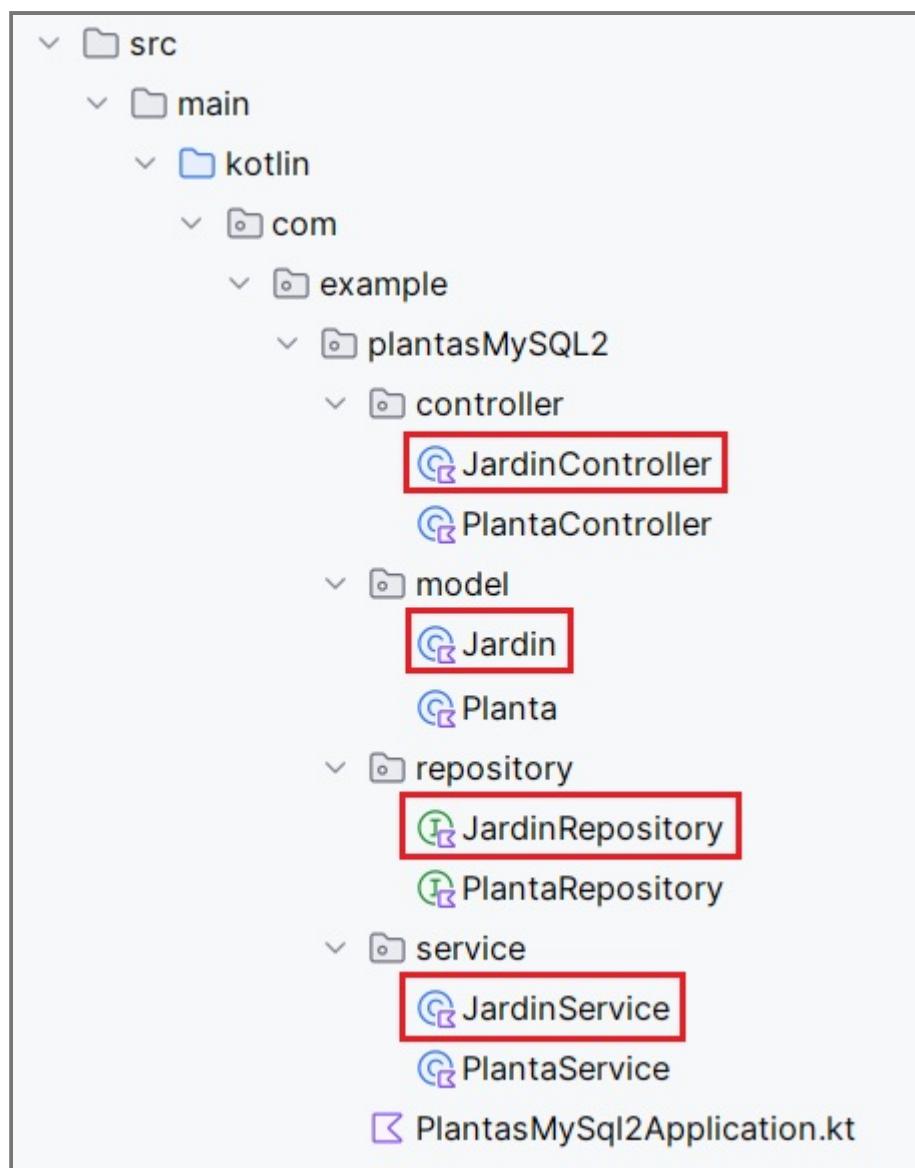
Para añadir la gestión de los jardines debemos añadir a la BD una nueva tabla llamada `jardines` (insertamos dos registros):

```
CREATE TABLE jardines (
    id_jardin INT AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(255) NOT NULL,
    ubicacion VARCHAR(255)
) ENGINE=InnoDB;

INSERT INTO jardines (nombre, ubicacion)
VALUES ('Botánico', 'Madrid');

INSERT INTO jardines (nombre, ubicacion)
VALUES ('Tropical', 'Valencia');
```

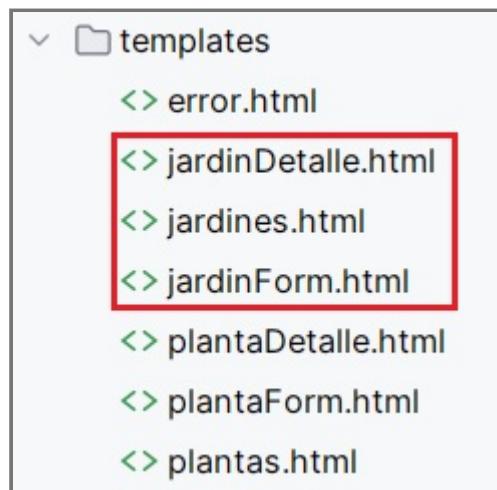
Luego añadimos a la aplicación los archivos necesarios para las operaciones CRUD sobre los jardines (equivalentes a los que tenemos para las plantas):



Por último añadimos los archivos `html` (también equivalentes a los que tenemos para las plantas). Si modificamos el archivo de error nos puede servir tanto para plantas como para jardines, su nuevo código es el siguiente:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Error</title>
    <link rel="stylesheet" th:href="@{/bootstrap/css/bootstrap.min.css}">
</head>
<body>
<div class="container mt-5 text-center">
    <h1 class="text-danger">Error</h1>
    <p>La información que estás buscando no existe.</p>
    <p><a href="/" class="mt-5">Volver a la pantalla de inicio</a></p>
</div>
</body>
</html>
```

De forma que los archivos `html` de nuestra aplicación son los siguientes:



PASO 3: Comprobar jardines

Al abrir <http://localhost:8080/jardines> en el navegador veremos lo siguiente:

ID	Nombre	Ubicación	Acciones
1	Botánico	Madrid	Detalles Editar Borrar
2	Tropical	Valencia	Detalles Editar Borrar

[Agregar Nuevo Jardín](#)

El funcionamiento es exactamente el mismo que el de las plantas.

Una vez tenemos las plantas y los jardines funcionando de forma independiente vamos a añadir una pantalla a nuestra aplicación para poder llevar plantas a los jardines. Para ello añadimos una nueva tabla a la base de datos con la relación (muchos a muchos) entre jardines y plantas y añadimos los archivos necesarios a nuestra aplicación.

PASO 4: Ampliar la base de datos

Añadir la tabla `jardines_plantas` a la base de datos y hacemos dos inserts (en este caso suponemos que existen los jardines con id 1 y 2 y las plantas con id 1 y 2:

```
CREATE TABLE jardines_plantas (
    id_jardin INT NOT NULL,
    id_planta INT NOT NULL,
    cantidad INT NOT NULL,
    PRIMARY KEY (id_jardin, id_planta)
) ENGINE=InnoDB;

INSERT INTO jardines_plantas (id_jardin, id_planta, cantidad)
VALUES (1, 1, 10);

INSERT INTO jardines_plantas (id_jardin, id_planta, cantidad)
VALUES (2, 2, 5);
```

PASO 5: Añadir el modelo

En este caso el modelo es un poco más complejo ya que necesita tener configurada la relación muchos a muchos entre las plantas y los jardines. Creamos el archivo `JardinPlanta.kt` dentro de la carpeta `src/main/kotlin/com/example/plantasMySQL2/model/` con el siguiente código:

```
package com.example.plantasMySQL2.model
import jakarta.persistence.*

@Embeddable
class JardinPlantaId(
    var idJardin: Int = 0,
    var idPlanta: Int = 0
)

@Entity
@Table(name = "jardines_plantas")
class JardinPlanta(

    @EmbeddedId
    var id: JardinPlantaId = JardinPlantaId(),

    @ManyToOne
    @MapsId("idJardin")
    @JoinColumn(name = "id_jardin")
    var jardin: Jardin? = null,

    @ManyToOne
    @MapsId("idPlanta")
    @JoinColumn(name = "id_planta")
    var planta: Planta? = null,

    @Column(nullable = false)
    var cantidad: Int = 0
)
```

PASO 6: Añadir el repositorio

De momento el repositorio no contendrá código, lo ampliaremos más adelante, ahora creamos el archivo `JardinPlantaRepository.kt` dentro de la carpeta `src/main/kotlin/com/example/plantasMySQL2/repository/` con el código siguiente:

```
package com.example.plantasMySQL2.repository

import com.example.plantasMySQL2.model.JardinPlanta
import com.example.plantasMySQL2.model.JardinPlantaId
import org.springframework.data.jpa.repository.JpaRepository
import org.springframework.stereotype.Repository

import org.springframework.data.jpa.repository.Query
import org.springframework.data.repository.query.Param

@Repository
interface JardinPlantaRepository : JpaRepository<JardinPlanta, JardinPlantaId> {}
```

PASO 7: Añadir la clase del servicio intermedio

Crear el archivo `JardinPlantaService.kt` dentro de la carpeta `src/main/kotlin/com/example/plantasMySQL2/service/` con el siguiente código:

```
package com.example.plantasMySQL2.service
```

```

import com.example.plantasMySQL2.model.JardinPlanta
import com.example.plantasMySQL2.model.JardinPlantaId
import com.example.plantasMySQL2.model.Jardin
import com.example.plantasMySQL2.model.Planta
import com.example.plantasMySQL2.repository.JardinPlantaRepository
import com.example.plantasMySQL2.repository.PlantaRepository
import com.example.plantasMySQL2.repository.JardinRepository
import org.springframework.stereotype.Service

@Service
class JardinPlantaService(
    private val jardinPlantaRepository: JardinPlantaRepository,
    private val jardinRepository: JardinRepository,
    private val plantaRepository: PlantaRepository
) {
    fun listarTodas(): List<JardinPlanta> =
        jardinPlantaRepository.findAll()

    // Métodos para llenar los desplegables
    fun listarJardines(): List<Jardin> = jardinRepository.findAll()
    fun listarPlantas(): List<Planta> = plantaRepository.findAll()

    // Método para guardar
    fun guardar(idJardin: Int, idPlanta: Int, cantidad: Int) {
        // 1. Buscamos las entidades (lanzará error si no existen, lo cual es bueno para integridad)
        val jardinRef = jardinRepository.findById(idJardin).orElseThrow()
        val plantaRef = plantaRepository.findById(idPlanta).orElseThrow()

        // 2. Creamos la clave compuesta
        val id = JardinPlantaId(idJardin, idPlanta)

        // 3. Creamos la entidad relación
        val nuevaRelacion = JardinPlanta(
            id = id,
            jardin = jardinRef,
            planta = plantaRef,
            cantidad = cantidad
        )

        jardinPlantaRepository.save(nuevaRelacion)
    }
}

```

PASO 8: Añadir el controlador

Crear el archivo `JardinPlantaController.kt` dentro de la carpeta `src/main/kotlin/com/example/plantasMySQL2/controller/` con el siguiente código:

```

package com.example.plantasMySQL2.controller

import com.example.plantasMySQL2.model.JardinPlanta
import com.example.plantasMySQL2.model.Planta
import com.example.plantasMySQL2.service.JardinPlantaService
import org.springframework.stereotype.Controller
import org.springframework.ui.Model
import org.springframework.web.bind.annotation.*

// Clase auxiliar para capturar los datos del formulario
class JardinPlantaForm(
    var idJardin: Int = 0,
    var idPlanta: Int = 0,
    var cantidad: Int = 0
)

@Controller
class JardinPlantaController(
    private val jardinPlantaService: JardinPlantaService
) {
    @GetMapping("/jp")
    fun listaPlantasJardines(model: Model): String {
        val relaciones = jardinPlantaService.listarTodas()
        model.addAttribute("relaciones", relaciones)
        return "jp"
    }

    // Mostrar el formulario (GET)
    @GetMapping("/jp/add")
    fun mostrarFormulario(model: Model): String {
        // Pasamos un objeto vacío para el binding del formulario
        model.addAttribute("form", JardinPlantaForm())

        // Pasamos las listas para los desplegables
        model.addAttribute("jardines", jardinPlantaService.listarJardines())
        model.addAttribute("plantas", jardinPlantaService.listarPlantas())

        return "jpForm" // Nombre del archivo HTML nuevo
    }

    // Procesar el formulario (POST)
}

```

```

@PostMapping("/jp/guardar")
fun guardarRelacion(@ModelAttribute("form") form: JardinPlantaForm): String {
    jardinPlantaService.guardar(form.idJardin, form.idPlanta, form.cantidad)
    return "redirect:/jp" // Redirige a la lista principal
}
}

```

PASO 9: Añadir las vistas

En este caso añadiremos dos vistas, la primera mostrará una lista con los nombres de los jardines, los nombres de las plantas que tienen y la cantidad de éstas en el jardín. La segunda vista será un formulario para añadir una planta a un jardín de forma que si no existe la relación en la base de datos se añadirá un registro y si existe se actualizará la cantidad.

El código del archivo `jp.html` dentro de la carpeta `src/main/resources/templates/` es el siguiente:

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Informe Plantas por Jardín</title>
    <link rel="stylesheet" th:href="@{/bootstrap/css/bootstrap.min.css}">
</head>

<body>
<div class="container mt-5">

    <h1>Ubicación de plantas en jardines</h1>

    <h5 th:if="${relaciones.size() > 0}">
        Plantas, jardines y cantidades
    </h5>
    <p th:unless="${relaciones.size() > 0}">
        No hay datos para mostrar
    </p>

    <table class="table table-striped mt-4" th:if="${relaciones.size() > 0}">
        <thead>
            <tr>
                <th>Planta</th>
                <th>Jardín</th>
                <th>Cantidad</th>
            </tr>
        </thead>
        <tbody>
            <tr th:each="jp : ${relaciones}">
                <td th:text="${jp.planta.nombre}">Rosa</td>
                <td th:text="${jp.jardin.nombre}">Jardín Botánico</td>
                <td th:text="${jp.cantidad}">5</td>
            </tr>
        </tbody>
    </table>

    <a th:href="@{/jp/add}" class="btn btn-secondary btn-lg">Agregar planta a jardín</a>

    <p><a href="/" class="mt-5">Volver a la pantalla de inicio</a></p>
</div>
</body>
</html>

```

El código del archivo `jpForm.html` dentro de la carpeta `src/main/resources/templates/` es el siguiente:

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Asignar Planta a Jardín</title>
    <link rel="stylesheet" th:href="@{/bootstrap/css/bootstrap.min.css}">
</head>
<body>
<div class="container mt-5">
    <form th:action="@{/jp/guardar}" th:object="${form}" method="post">

        <div class="mb-3">
            <label for="jardin" class="form-label">Selecciona el Jardín:</label>
            <select class="form-select" id="jardin" name="idJardin">
                <option value="" disabled selected>-- Elegir Jardín --</option>
                <option th:each="jardin : ${jardines}"
                    th:value="${jardin.id_jardin}"
                    th:text="${jardin.nombre}"></option>
            </select>
        </div>

        <div class="mb-3">
            <label for="planta" class="form-label">Selecciona la Planta:</label>
            <select class="form-select" id="planta" name="idPlanta">
                <option value="" disabled selected>-- Elegir Planta --</option>
                <option th:each="planta : ${plantas}"></option>
            </select>
        </div>
    </form>
</div>

```

```

        th:value="${planta.id_planta}"
        th:text="${planta.nombre}"></option>
    </select>
</div>

<div class="mb-3">
    <label for="cantidad" class="form-label">Cantidad:</label>
    <input type="number" class="form-control" id="cantidad"
           th:field="*{cantidad}" min="1" required>
</div>

<div class="d-grid gap-2">
    <button type="submit" class="btn btn-success">Guardar</button>
    <a th:href="@{/jp}" class="btn btn-secondary">Cancelar</a>
</div>
</form>
</div>
</body>
</html>
```

PASO 10: Consulta @Query

En este último paso, añadiremos una nueva pantalla que devolverá las plantas (y sus cantidades) que hay en un jardín concreto. Para ello, añadimos un botón a la pantalla de Ubicación de plantas en jardines que nos llevará a una pantalla con un desplegable para elegir el jardín. La consulta se realizará utilizando `@Query`. A continuación se muestra el código nuevo:

- En el archivo `jp.html` añadimos el código para el nuevo botón:

```
<a th:href="@{/informe/plantas-por-jardin}" class="btn btn-primary mt-3">
    Filtrar por jardín </a>
```

- En `JardinPlantaController.kt` añadimos el código del formulario y del resultado de la búsqueda:

```

// Mostrar el formulario con el desplegable
@GetMapping("//jp/plantas-por-jardin")
fun formularioDePlantasPorJardin(model: Model): String {
    model.addAttribute("jardines", jardinService.listar())
    return "plantasPorJardinForm"
}

// Procesar el jardín seleccionado
@PostMapping("/jp/plantas-por-jardin")
fun mostrarPlantasDeJardin(
    @RequestParam idJardin: Int,
    model: Model
): String {

    val relaciones = jardinPlantaService.obtenerPlantasDeJardin(idJardin)
    model.addAttribute("relaciones", relaciones)
    model.addAttribute("jardines", jardinService.listar())
    model.addAttribute("idJardinSeleccionado", idJardin)

    return "plantasPorJardinForm"
}
```

- En `JardinPlantaRepository.kt` añadimos la consulta:

```

@Query("""
    SELECT jp
    FROM JardinPlanta jp
    JOIN jp.planta p
    WHERE jp.jardin.id_jardin = :idJardin
    ORDER BY p.nombre
""")
fun obtenerPlantasDeJardin(
    @Param("idJardin") idJardin: Int
): List<JardinPlanta>
```

- En `JardinPlantaService.kt` añadimos el código que llama a la consulta:

```
fun obtenerPlantasDeJardin(idJardin: Int): List<JardinPlanta> =
    jardinPlantaRepository.obtenerPlantasDeJardin(idJardin)
```

- Creamos la vista `plantasPorJardinForm.html` con el formulario y el resultado de la consulta:

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Plantas por jardín</title>
    <link rel="stylesheet" th:href="@{/bootstrap/css/bootstrap.min.css}">
```

```

</head>
<body>
<div class="container mt-5">
    <h1>Plantas por jardín</h1>
    <form th:action="@{/jp/plantas-por-jardin}" method="post" class="row g-3 mb-4">
        <div class="col-md-6">
            <label class="form-label">Selecciona un jardín:</label>
            <select name="idJardin" class="form-select" required>
                <option value="">-- Selecciona --</option>
                <option th:each="j : ${jardines}"
                    th:value="${j.id_jardin}"
                    th:text="${j.nombre}"
                    th:selected="${j.id_jardin == idJardinSeleccionado}">
                </option>
            </select>
        </div>
        <div class="col-md-6 align-self-end">
            <button type="submit" class="btn btn-success">
                Mostrar plantas
            </button>
        </div>
    </form>
    <table class="table table-striped" th:if="${relaciones != null}">
        <thead>
            <tr>
                <th>Planta</th>
                <th>Cantidad</th>
            </tr>
        </thead>
        <tbody>
            <tr th:each="jp : ${relaciones}">
                <td th:text="${jp.planta.nombre}">Rosa</td>
                <td th:text="${jp.cantidad}">5</td>
            </tr>
        </tbody>
    </table>
    <a th:href="@{/jp}" class="btn btn-secondary">Volver</a>
</div>
</body>
</html>

```

✓ Prueba y analiza el ejemplo 5

1. Crea un proyecto Spring Boot llamado `plantasMySQL2` utilizando Spring Initializr.
2. Prueba el código del ejemplo, verifica que funciona correctamente y pregunta tus dudas.

⚠️ Práctica 3: Amplía tu aplicación

Amplía tu aplicación para que cumpla:

1. Tener dos tablas independientes y una tercera tabla que las relaciones de muchos a muchos.
2. CRUD de las 3 tablas.
3. Pantalla que funcione de forma similar a la del ejemplo utilizando una consulta @Query

 **Entrega**

Realiza lo siguiente:

1. Exporta tu BD a un archivo .sql
2. Comprime la carpeta `main` de tu proyecto en formato .zip
3. Entrega en Aules el archivo .sql y el archivo .zip

IMPORTANTE: El proyecto no debe contener código que no se utilice, ni restos de pruebas de los ejemplos y no debe estar separado por prácticas. Debe ser un proyecto totalmente funcional.

Autoría

Obra realizada por Begoña Paterna Lluch basada en materiales desarrollados por Alicia Salvador Contreras. Publicada bajo licencia [Creative Commons Atribución/Reconocimiento-CompartirIgual 4.0 Internacional](#)

3. Material adicional

3.1 Ubuntu Server en AWS Learner Lab

Revisiones

Revisión	Fecha	Descripción
1.0	11-10-2025	Adaptación de los materiales a markdown
1.1	21-10-2025	Reorganización de algunas secciones
1.2	06-11-2025	Ampliación sección de exportación de la BD
1.3	10-11-2025	Eliminación del apartado referente a MySQL

Acceso al laboratorio

A continuación se describen los pasos para acceder al laboratorio de aprendizaje de AWS Academy.

1. Crea tu cuenta

Habráis recibido un correo electrónico de invitación, haz clic en el enlace y crea tu cuenta. Una vez completado el registro se abrirá el entorno de AWS Academy automáticamente. Haz clic en [Contenidos](#) y luego en el enlace [Lanzamiento del laboratorio](#) como se muestra en la siguiente imagen (la primera vez que entres deberás aceptar los términos de uso)

The screenshot shows the AWS Academy course interface. On the left, there's a sidebar with various navigation options: Página de inicio, Contenidos (which is selected and highlighted with a red border), Foros de discusión, Notas, Lucid (pizarra), Asignaturas, Calendario, Bandeja de entrada, Historial, and Ayuda. The main content area is titled 'ALLv2ES-ES-LTI13-141646 > Contenidos'. It displays several course modules:

- Bienvenida e información general sobre el curso**
 - Encuesta previa al curso [Pre-Courise Survey ES-ES]**
 - Guía del alumno del Laboratorio para el alumnado de AWS Academy**
- Conformidad y seguridad del Laboratorio para el alumnado de AWS Academy**
 - Aprende a utilizar eficazmente el Laboratorio para el alumnado de Academy**
- Módulo Prueba de conocimientos**

100 puntos Puntuación mínima 70.0
- Laboratorio para el alumnado de AWS Academy**
 - Lanzamiento del Laboratorio para el alumnado de AWS Academy**

Si te aparece el siguiente mensaje: "This assignment is locked till you access it through your respective LMS once, please use your LMS to access/unlock this assignment" vuelve a hacer clic en *Contenidos* y en el enlace *Lanzamiento del laboratorio*

Para entrar al curso en el futuro hazlo desde: https://www.awsacademy.com/vforcesite/LMS_Login



Luego haz clic en el nombre del curso y, una vez dentro, haz clic en `Contenidos` y en el enlace `Lanzamiento del laboratorio` como hiciste la primera vez

Complimentary access to AWS Skill Builder for 12 months

Great news! The AWS Academy Skill Builder free 12-month subscription access and certification vouchers initiative is now open to everyone. To activate the offer, please look in one of your active AWS Academy Courses. Select the modules menu item. At the bottom, under AWS T&C Resources, you will see a link for FAQs and the activation link to start the offer process. After you submit the form, you will be invited via email to AWS Skill Builder within 3 business days.

Este es un mensaje de AWS Academy

Notificaciones. Cuéntanos cómo y cuándo te gustaría recibir notificaciones de los eventos en Canvas.

Preferencias de notificación

Panel de control

AWS Academy Learner Lab [1416...]
ALLv2ES-ES-LTI13-141646

ALLv2ES-ES-LTI13-141646 > Contenidos

- Página de inicio
- Contenidos**
- Foros de discusión
- Notas
- Lucid (pizarra)

Bienvenida e información general sobre el curso

- Encuesta previa al curso [Pre-Couirse Survey ES-ES]
- Guía del alumno del Laboratorio para el alumnado de AWS Academy

Conformidad y seguridad del Laboratorio para el alumnado de AWS Academy

- Aprende a utilizar eficazmente el Laboratorio para el alumnado de Academy

Módulo Prueba de conocimientos
100 puntos Puntuación mínima 70.0

Laboratorio para el alumnado de AWS Academy

Lanzamiento del Laboratorio para el alumnado de AWS Academy

2. Inicia el laboratorio

Cuando aparezca la pantalla con el laboratorio, haz clic en el botón `Start Lab` (verás que el círculo junto al enlace `AWS` cambia de color rojo a amarillo y permanece de ese color mientras arranca el laboratorio)

The screenshot shows the AWS Learner Lab interface. At the top, there's a navigation bar with the URL `ALLv2ES-ES-LTI13-141646 > Contenidos > Laboratorio para el alumnado de AWS Academy > Lanzamiento del Laboratorio para el alumnado de AWS Academy`. Below the navigation bar, there's a sidebar with links: `Página de inicio`, `Contenidos` (which is selected), `Foros de discusión`, `Notas`, and `Lucid (pizarra)`. The main area features a large blue downward-pointing arrow with a red curved arrow at its top. In the top right corner of this area, there's a red box highlighting the `AWS` button. To the right of the main area, there's a sidebar titled `Learner Lab` with a dropdown menu set to `EN-US`. The sidebar contains links to various AWS services and resources, such as `Environment Overview`, `Environment Navigation`, `Access the AWS Management Console`, `Region restriction`, `Service usage and other restrictions`, `Using the terminal in the browser`, `Running AWS CLI commands`, `Using the AWS SDK for Python`, `Preserving your budget`, `Accessing EC2 Instances`, `SSH Access to EC2 Instances`, `SSH Access from Windows`, and `SSH Access from a Mac`. Below these links, it says `Instructions last updated: 2025-06-24`. At the bottom of the sidebar, under `Environment Overview`, it states: `This Learner Lab provides a sandbox environment for ad-hoc exploration of AWS services.`

3. Accede a la consola

Cuando el laboratorio haya arrancado, el círculo cambiará a color verde. Entonces haz clic en el enlace `AWS` para acceder a la `Página de inicio de la Consola` (puedes ver que la región es *North Virginia (us-east-1)* que es la región por defecto de los laboratorios de aprendizaje, **NO LO CAMBIAS**). Despues haz clic en `EC2` para acceder a la consola de instancias EC2

The screenshot shows the AWS Home Page. On the left, under 'Recent Visits', there is a section titled 'Servicios no visitados recientemente' with a red box around the 'EC2' button. Below this, there is a link to 'Ver todos los servicios'. On the right, under 'Aplicaciones', it shows 'Región: US East (N. Virginia)' with a red box around the dropdown menu set to 'us-east-1 (Región actual)'. There is also a search bar for 'Buscar aplicaciones'.

Instalación del servidor

A continuación se describen los pasos para crear un servidor Ubuntu en un laboratorio de aprendizaje de AWS Academy.

1. Crea la instancia

Dentro del panel EC2 haz clic en el botón **Lanzar la instancia**. Luego Escribe el nombre de la instancia y elige una Amazon Machine Image (AMI) en este caso Ubuntu (al seleccionar ubuntu nos aparece la Ubuntu Server 24.04 LTS que es apta para utilizar de forma gratuita). Más abajo también aparece el tipo de instancia que es

The screenshot shows the AWS EC2 console dashboard. On the left, there's a sidebar with sections like 'Panel', 'Instancias', 'Imagenes', and 'Elastic Block Store'. The main area has a 'Recursos' section with tables for instances, capacity reservations, auto-scaling groups, locations, instances, key pairs, load balancers, elastic IPs, security groups, dedicated hosts, snapshots, and volumes. To the right, there are sections for 'Atributos de la cuenta' (VPC predeterminada, Configuración), 'Información adicional' (Instrucciones para comenzar, Documentación, etc.), and 'Estado del servicio' (Panel de AWS Health). A red box highlights the 'Lanzar la instancia' button in the 'Lanzar la instancia' section.

This screenshot shows the first step of the 'Lanzar una instancia' wizard. It's titled 'Nombre y etiquetas'. It includes a 'Nombre' input field with a placeholder 'Indica el nombre de tu instancia' (highlighted with a red arrow), an 'Agregar etiquetas adicionales' link, and a 'Información' link. Below this is a section for 'Imagenes de aplicaciones y sistemas operativos (Imagen de máquina de Amazon)' with a search bar and a list of operating system icons: Amazon Linux, macOS, Ubuntu (highlighted with a red box), Windows, Red Hat, SUSE Linux, and Debian. A 'Buscar más AMI' link and a note about including AWS Marketplace and community AMIs are also present. At the bottom, there's a table for 'Ubuntu Server 24.04 LTS (HVM), SSD Volume Type' with details like AMI ID, virtualization type, and device type, along with an 'Apto para la capa gratuita' dropdown.

▼ **Tipo de instancia** [Información](#) | [Obtener asesoramiento](#)

Tipo de instancia

t3.micro Apto para la capa gratuita

Familia: t3 2 vCPU 1 GiB Memoria Generación actual: true
Bajo demanda Ubuntu Pro base precios: 0.0139 USD por hora
Bajo demanda SUSE base precios: 0.0104 USD por hora
Bajo demanda Linux base precios: 0.0104 USD por hora
Bajo demanda RHEL base precios: 0.0392 USD por hora
Bajo demanda Windows base precios: 0.0196 USD por hora

Todas las generaciones

[Comparar tipos de instancias](#)

Se aplican costos adicionales a las AMI con software preinstalado

2. Crea un par de claves

Haz clic en el enlace [Crear un nuevo par de claves](#), introduce el nombre para el fichero de claves y haz clic en el botón [Crear par de claves](#)

▼ **Par de claves (inicio de sesión)** [Información](#)

Puede utilizar un par de claves para conectarse de forma segura a la instancia. Asegúrese de que tiene acceso al par de claves seleccionado antes de lanzar la instancia.

Nombre del par de claves - obligatorio

▼ C [Crear un nuevo par de claves](#)

Crear par de claves

Nombre del par de claves
Con los pares de claves es posible conectarse a la instancia de forma segura.

← Nombre de tu clave

El nombre puede incluir hasta 255 caracteres ASCII. No puede incluir espacios al principio ni al final.

Tipo de par de claves

- RSA
Par de claves pública y privada cifradas mediante RSA
- ED25519
Par de claves privadas y públicas cifradas ED25519

Formato de archivo de clave privada

- .pem
Para usar con OpenSSH
- .ppk
Para usar con PuTTY

⚠ Cuando se le solicite, almacene la clave privada en un lugar seguro y accesible del equipo. Lo necesitará más adelante para conectarse a la instancia. [Más información](#) [i]

[Cancelar](#) Crear par de claves

Muy importante: Verás que el navegador descarga el fichero .pem de tu clave automáticamente. Guárdalo en lugar seguro porque te hará falta para conectar a tu servidor por SSH.

3. Lanzar instancia

Deja el resto de opciones como están y, en la parte derecha dentro del apartado **Resumen**, haz clic en el botón **Lanzar instancia**. Cuando la instancia termine de lanzarse aparecerá la siguiente imagen

Correcto
El lanzamiento de la instancia se inició correctamente (i-052a4eee6a6a8f916)

▶ Registro de lanzamiento

Pasos siguientes

- Crea alertas de uso de facturación**
Para controlar los costos y evitar cargos inesperados, configure notificaciones por correo electrónico que avisen cuando se alcancen ciertos umbrales de uso.
[Crear alertas de facturación](#)
- Conectarse a la instancia**
Una vez que la instancia esté en ejecución, inicie sesión en ella desde el equipo local.
[Conectarse a la instancia](#)
[Más información](#)
- Conectar una base de datos de RDS**
Configure la conexión entre una instancia de EC2 y una base de datos para permitir el flujo de tráfico entre ellas.
[Conectar una base de datos de RDS](#)
[Crear una nueva base de datos de RDS](#)
[Más información](#)

[Ver todas las instancias](#)

4. Asignar una IP pública fija

Para que el servidor tenga siempre la misma IP pública y facilitar así trabajar con él, vamos a asignarle una IP fija. Para ello haz clic en el enlace `Direcciones IP elásticas` y luego haz clic en el botón `Asignar dirección IP elástica`.

Recursos

Actualmente, utiliza los siguientes recursos de Amazon EC2 en la región Estados Unidos (Norte de Virginia):

Instancias (en ejecución)	1	Balanceadores de carga	0	Capacity Reservations	0
Direcciones IP elásticas	0	Grupos de escalamiento automático	0	Grupos de seguridad	3
Grupos de ubicación	0	Hosts dedicados	0	Instancias	1
Instantáneas	0	Pares de claves	3	Volúmenes	1

Lanzar la instancia

Para comenzar, lance una instancia de Amazon EC2, que es un servidor virtual en la nube.

Estado del servicio

Panel de AWS Health

Zonas

Nombre de la zona	ID de la zona
us-east-1a	use1-az1
us-east-1b	use1-az2
us-east-1c	use1-az4
us-east-1d	use1-az6
us-east-1e	use1-az3
us-east-1f	use1-az5

The screenshot shows the AWS Management Console interface for the EC2 service, specifically the 'Direcciones IP elásticas' (Elastic IP Addresses) section. The left sidebar contains navigation links for EC2 services like Instances, Images, Elastic Block Store, Network & Security, and Load Balancing. The main content area is titled 'Direcciones IP elásticas' and displays a message: 'No se encontraron direcciones IP elásticas en esta región'. At the top right of this area, there is a blue 'Acciones' (Actions) button and an orange 'Asignar dirección IP elástica' (Assign Elastic IP Address) button, which is highlighted with a red box. Below this, a modal window titled 'Seleccionar una dirección IP elástica' (Select an Elastic IP Address) provides instructions: 'Vea el uso de las direcciones IP y las recomendaciones para liberar las IP no utilizadas con Public IP insights.' (View the use of IP addresses and recommendations to release unused IPs with Public IP insights.)

En la siguiente pantalla deja las opciones por defecto y haz clic en el botón **Asignar**. Verás la nueva IP

The screenshot shows the AWS Management Console interface for assigning an elastic IP address. At the top, there's a navigation bar with the AWS logo, a search bar, and various account and region settings. Below the navigation, the breadcrumb trail indicates the user is in the EC2 service under 'Direcciones IP elásticas' (Elastic IP Addresses) and is currently on the 'Asignar dirección IP elástica' (Assign Elastic IP Address) page.

Configuraciones de la dirección IP elástica Información

Grupo de direcciones IPv4 públicas

- El grupo de direcciones IPv4 de Amazon
- Dirección IPv4 pública que trae a su cuenta de AWS con BYOIP. (opción deshabilitada porque no se encontraron grupos) [Más información](#)
- Grupo de direcciones IPv4 propiedad del cliente creado desde su red local para su uso con un Outpost. (opción deshabilitada porque no se encontró ningún grupo propiedad del cliente) [Más información](#)
- Asignar mediante un grupo de IPAM de IPv4 (opción deshabilitada porque no se encontró ningún grupo público de IPAM de IPv4 con un servicio de AWS como EC2)

Grupo fronterizo de red Información

Search input: us-east-1 X

Direcciones IP estáticas globales

AWS Global Accelerator puede proporcionar direcciones IP estáticas globales que se anuncian en todo el mundo mediante anycast desde ubicaciones periféricas de AWS. Esto puede ayudar a mejorar la disponibilidad y la latencia del tráfico de usuarios mediante el uso de la red global de Amazon. [Más información](#)

[Crear un acelerador](#)

Etiquetas - opcional

Una etiqueta es una marca que se asigna a un recurso de AWS. Cada etiqueta se compone de una clave y un valor opcional. Puede usar etiquetas para buscar y filtrar sus recursos o realizar un seguimiento de los costos de AWS.

No hay etiquetas asociadas al recurso.

[Agregar una etiqueta nueva](#)

Puede agregar hasta 50 etiquetas más

[Cancelar](#) Asignar

The screenshot shows the AWS Management Console with the EC2 service selected. In the left sidebar, under the 'Red y seguridad' section, 'Direcciones IP elásticas' is selected. The main pane displays a table of elastic IP addresses. A message box at the top right indicates that the IP '100.25.102.165' was assigned successfully. The assigned IP is highlighted with a red box in the 'Dirección IPv4 asignada' column.

Name	Dirección IPv4 asignada	Tipo
-	100.25.102.165	IP pública

Selecciona la nueva IP, haz clic en el desplegable **Acciones** y entra en **Dirección IP elástica asociada**, luego indica la instancia a la que asignar la IP y haz clic en **Asociado**. La IP quedará asociada a tu instancia

Screenshot 1: AWS EC2 - Asociar dirección IP elástica

La dirección IP elástica se asignó correctamente. Dirección IP elástica 100.25.102.165

Acciones (2) Asignar dirección IP elástica (3)

Dirección IP elástica asociada (1)

Screenshot 2: AWS EC2 - Asociar dirección IP elástica

Dirección IP elástica asociada

Instancia: i-0193981641dd0ccbf

Reasociación: Permitir que se vuelva a asociar esta dirección IP elástica

Botones: Cancelar (Azul) Asociado (Amarillo)

The screenshot shows the AWS EC2 'Derecciones IP elásticas' (Elastic IP Addresses) page. A success message at the top states: 'La dirección IP elástica se asoció correctamente. La dirección IP elástica 100.25.102.165 se ha asociado a instancia i-0193981641dd0ccbf'. Below this, the IP address 100.25.102.165 is highlighted in a red box. The main table displays the following details:

Resumen	
Dirección IPv4 asignada 100.25.102.165	Tipo IP pública
ID de asociación eipassoc-0950095dffe375dc2	Ámbito VPC
ID de interfaz de red eni-091cd2cdab4a21d24	ID de la cuenta del propietario de la interfaz de red 679480731783
Grupo de direcciones Amazon	Grupo fronterizo de red us-east-1
	DNS público ec2-100-25-102-165.compute-1.amazonaws.com
	Service managed
	Registro DNS inverso -
	Dirección IP privada 172.31.20.80
	ID de puerta de enlace de NAT -

Vuelve a la lista de instancias para comprobar que tu instancia ya tiene la IP

The screenshot shows the AWS EC2 'Instancias' (Instances) page. The search bar contains 'Estado de la instancia = running'. The results table shows one instance:

<input checked="" type="checkbox"/> Name <input type="text" value="bpl"/>	<input checked="" type="checkbox"/> ID de la instancia <input type="text" value="i-0193981641dd0ccbf"/>	<input checked="" type="checkbox"/> Estado de la i... <input type="text" value="En ejecución"/>	<input checked="" type="checkbox"/> Tipo de inst... <input type="text" value="t3.micro"/>	<input checked="" type="checkbox"/> Comprobaci... <input type="text" value="3/3 compi"/>
---	---	---	---	--

The instance details for 'i-0193981641dd0ccbf (bpl)' are shown in the 'Detalles' tab. The 'Resumen de instancia' section includes:

- ID de la instancia: i-0193981641dd0ccbf
- Dirección IPv4 pública: 100.25.102.165 (highlighted in red)
- Dirección IPv6: -
- Estado de la instancia: En ejecución

Other sections visible include 'Estado y alarmas', 'Monitoreo', 'Seguridad', 'Redes', and 'Almacenamiento'.

Prueba de conexión

1. Ver opciones de conexión

Haz clic en el botón Conectarse a la instancia para ver las diferentes opciones de conexión y sus instrucciones, por ejemplo en la pestaña Cliente SSH (que es la que vamos a utilizar) aparece lo siguiente:

Conectar Información

Conéctese a una instancia a través del cliente basado en navegador.

Conexión de la instancia EC2 **Administrador de sesiones** **Cliente SSH** **Consola de serie de EC2**

ID de la instancia
i-052a4eee6a6a8f916 (plantas)

1. Abra un cliente SSH.
2. Localice el archivo de clave privada. La clave utilizada para lanzar esta instancia es claveAWS.pem
3. Ejecute este comando, si es necesario, para garantizar que la clave no se pueda ver públicamente.
 chmod 400 "claveAWS.pem"
4. Conéctese a la instancia mediante su DNS público:
 ec2-13-218-241-87.compute-1.amazonaws.com

Ejemplo: Nombre clave IP o nombre del servidor
 ssh -i <private_key>.pem ubuntu@<public_ip>

Nota: En la mayoría de los casos, el nombre de usuario adivinado es correcto. Sin embargo, lea las instrucciones de uso de la AMI para comprobar si el propietario de la AMI ha cambiado el nombre de usuario predeterminado de la AMI.

2. Conectar al servidor por ssh

Para conectar, abre una ventana de comandos y asegurate que el archivo .pem está en la carpeta desde la que lanzas el siguiente comando (puedes utilizar el nombre del servidor o su IP pública):

```
ssh -i [nombre_clave] ubuntu@[nombre_IP_servidor]
```

Si aparece el siguiente aviso:

```
@@@@@@@
@      WARNING: UNPROTECTED PRIVATE KEY FILE!      @
@@@@@@@
Permissions 0644 for '[REDACTED]' are too open.
It is required that your private key files are NOT accessible by others.
This private key will be ignored.
Load key "[REDACTED]": bad permissions
[REDACTED]: Permission denied (publickey).
```

Ejecuta el comando siguiente:

```
chmod 400 [nombre_clave]
```

Si la conexión se ha establecido correctamente verás la siguiente información:

```
C:\Users\bego>ssh -i claveAWS.pem ubuntu@ec2-13-218-241-87.compute-1.amazonaws.com
The authenticity of host 'ec2-13-218-241-87.compute-1.amazonaws.com (13.218.241.87)' can't be established.
ED25519 key fingerprint is SHA256:KOWlRme05dUE1zuNid9kLTybdLRL1LwUvzfES3dchdQ.
This host key is known by the following other names/addresses:
  C:\Users\bego/.ssh/known_hosts:9: 13.218.241.87
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'ec2-13-218-241-87.compute-1.amazonaws.com' (ED25519) to the list of known hosts.
Welcome to Ubuntu 24.04.3 LTS (GNU/Linux 6.14.0-1011-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro

System information as of Sat Oct 11 14:44:34 UTC 2025

System load: 0.0          Temperature:      -273.1 C
Usage of /:   26.0% of 6.71GB  Processes:       112
Memory usage: 25%          Users logged in:  0
Swap usage:   0%           IPv4 address for ens5: 172.31.16.226

Expanded Security Maintenance for Applications is not enabled.

0 updates can be applied immediately.

Enable ESM Apps to receive additional future security updates.
See https://ubuntu.com/esm or run: sudo pro status

The list of available updates is more than a week old.
To check for new updates run: sudo apt update

Last login: Sat Oct 11 14:34:26 2025 from 79.116.218.66
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

ubuntu@ip-172-31-16-226:~$
```

Autoría

Obra realizada por Begoña Paterna Lluch. Publicada bajo licencia [Creative Commons Atribución/Reconocimiento-CompartirIgual 4.0 Internacional](https://creativecommons.org/licenses/by-sa/4.0/)

3.2 DB Browser for SQLite

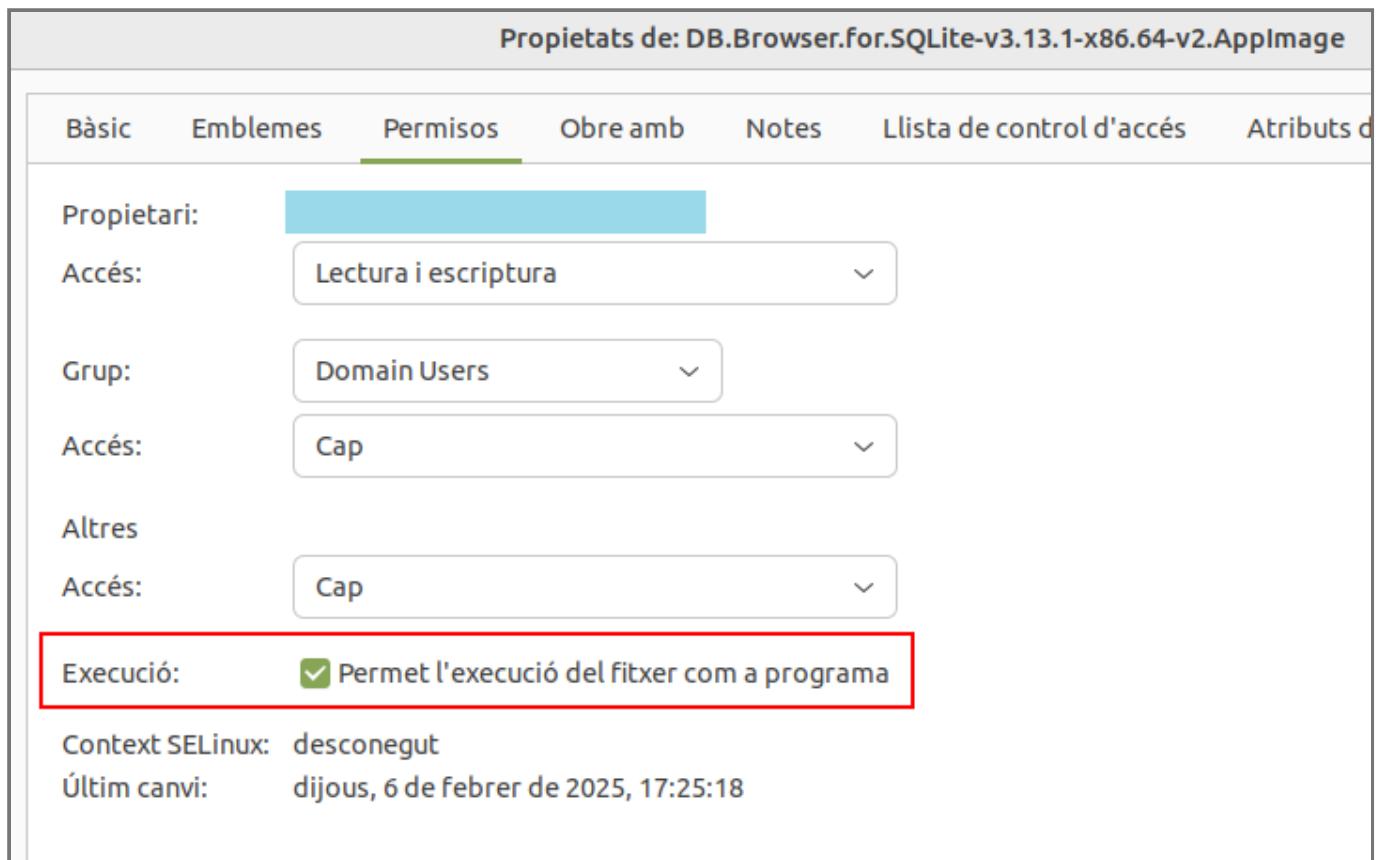
Siempre que desarrollamos aplicaciones que acceden a una base de datos es recomendable disponer de un visor / editor adecuado al sistema gestor de bases de datos con el que estamos trabajando para poder realizar comprobaciones. Con la herramienta DB Browser for SQLite podemos crear, modificar y consultar la estructura de una base de datos.

Se puede descargar desde el siguiente enlace <https://sqlitebrowser.org/dl/>

Para Linux puedes descargar la AppImage que no necesita instalación desde: <https://sqlitebrowser.org/dl/#linux>

Una vez descargada, para poder ejecutarla con doble clic, sigue estos pasos:

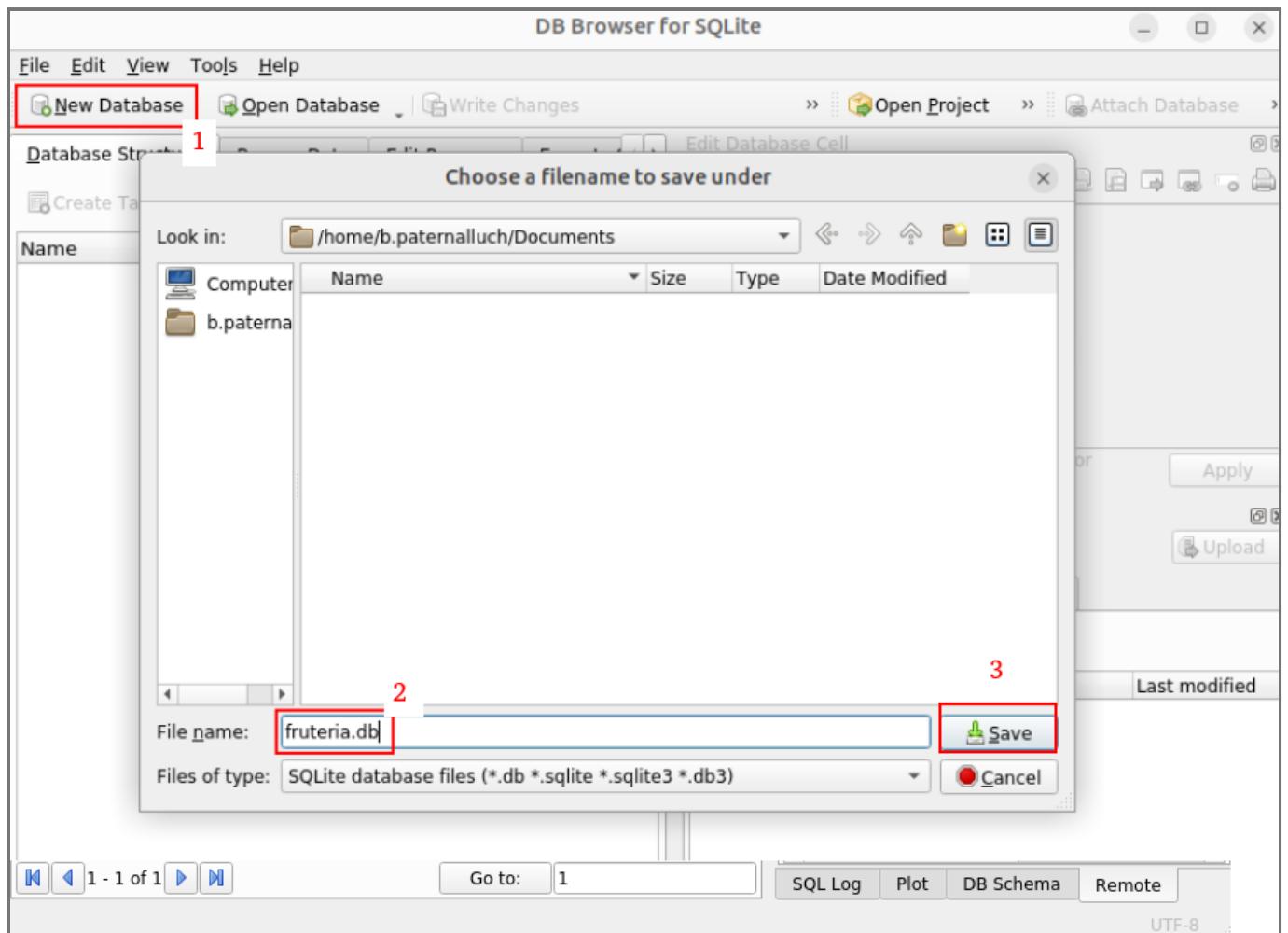
- Haz clic con el botón derecho y entra en Propiedades.
- Entra en la pestaña Permisos.
- Marca la casilla Permite la ejecución del fichero como programa.



A continuación se describen los pasos a seguir para crear una base de datos llamada **fruteria.db** con una tabla llamada **productos** con la siguiente estructura:

Campo	Tipo de Dato	Descripción
id	INTEGER (PK, AUTOINCREMENT)	Identificador único.
nombre	CHAR(15)	Nombre.
precio	REAL	Precio (con decimales).
procedencia	CHAR(3)	Código de país de procedencia.

Crear la base de datos



Crear la tabla y los campos

Edit table definition

Table **productos** 1

Advanced

Fields Index Constraints Foreign Keys Check Constraints

Add 2 Remove Move to top Move up Move down Move to bottom

Name	Type	NN	PK	AI	U	Default	Check	Collation
id	INTEGER	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>			
nombre	CHAR(15)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>			
precio	REAL	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>			
procedencia	CHAR(3)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>			

```

1 CREATE TABLE "productos" (
2     "id"      INTEGER,
3     "nombre"   CHAR(15),
4     "precio"    REAL,
5     "procedencia" CHAR(3),
6     PRIMARY KEY("id" AUTOINCREMENT)
7 );

```

4

Cancel OK

DB Browser for SQLite - /home/b.paternalluch/Documents/fruteria.db

File Edit View Tools Help

New Database Open Database Write Changes 4

Database Structure Browse Data Edit Pragmas Execute < >

Create Table Create Index Print Refresh

Tables (2)

- productos
- sqlite_sequence

Indices (0)

Views (0)

Triggers (0)

Edit Database Cell

Mode: Text

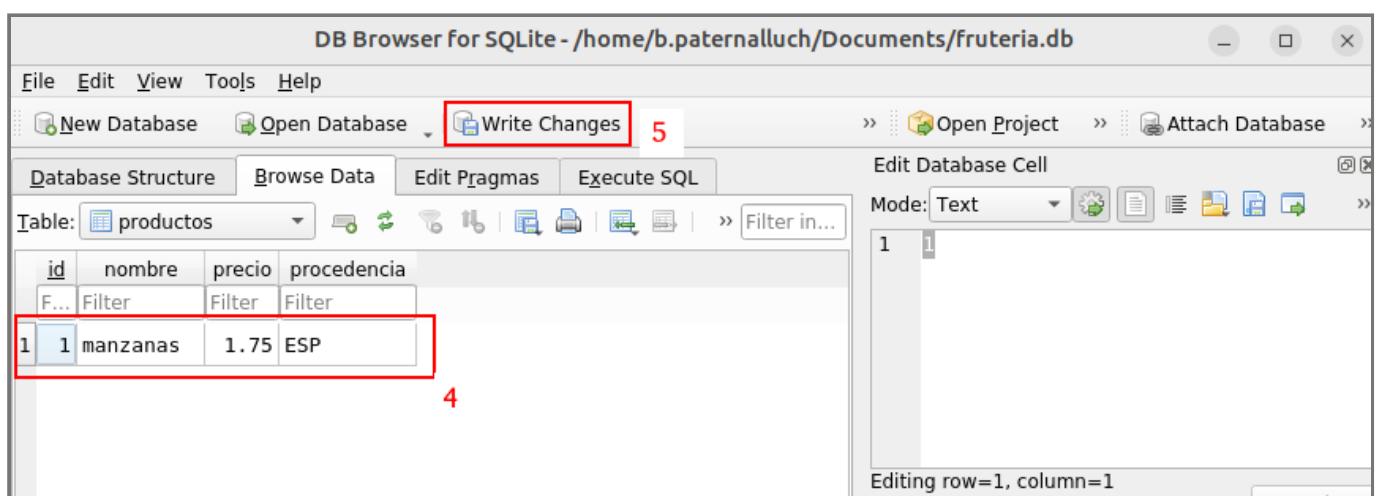
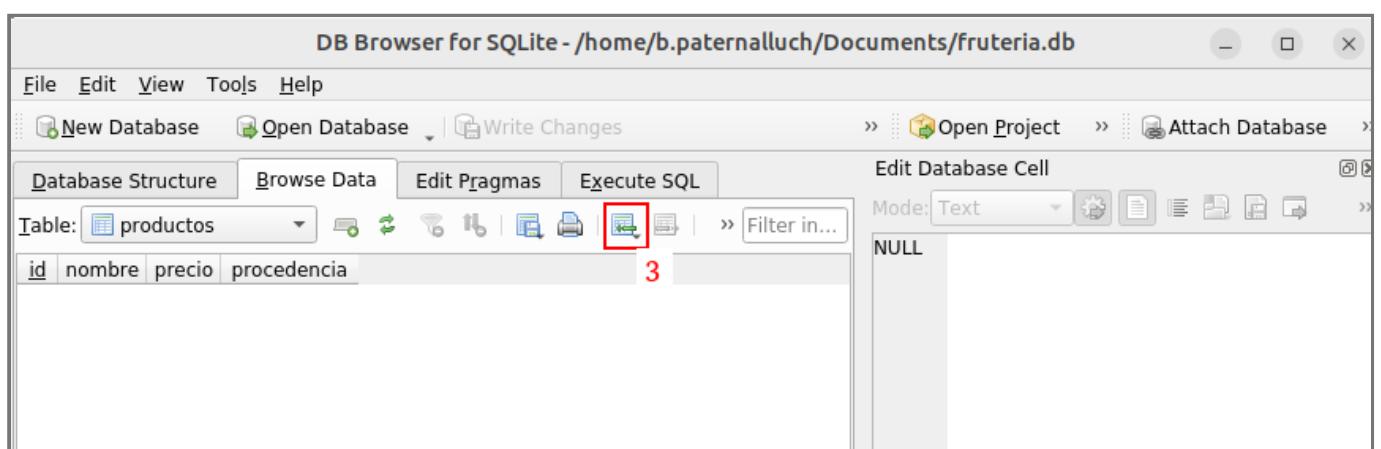
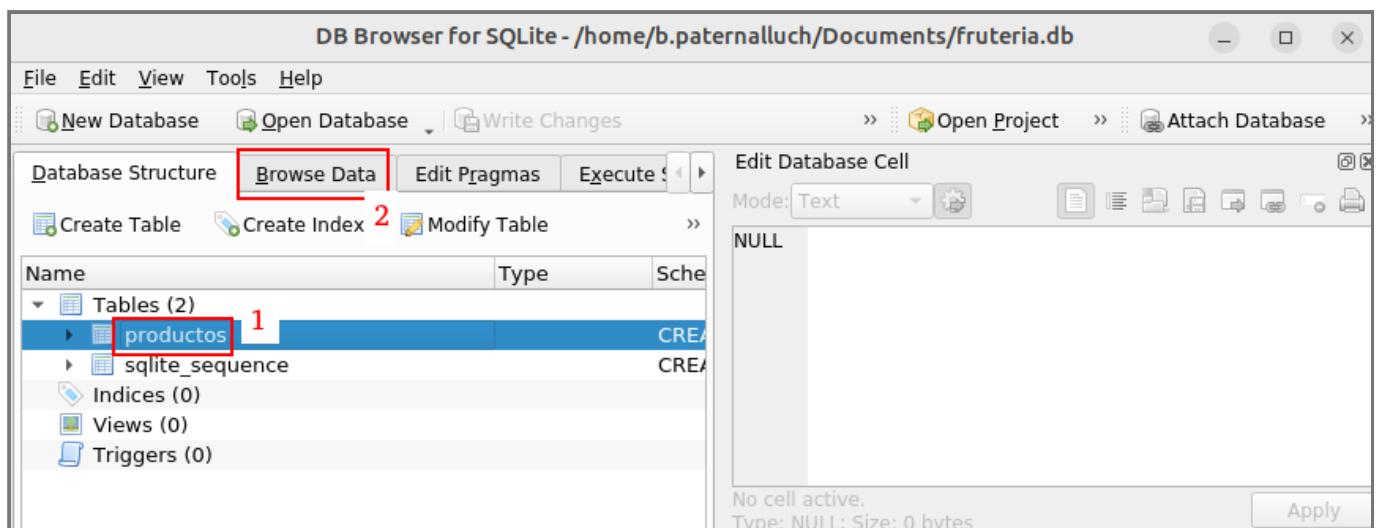
NULL

No cell active.

Type: NULL | Size: 0 bytes

Apply

Añadir información a la tabla



Autoría

Obra realizada por Begoña Paterna Lluch. Publicada bajo licencia [Creative Commons Atribución/Reconocimiento-CompartirIgual 4.0 Internacional](#)

3.3 DBeaver



Revisiones

Revisión	Fecha	Descripción
1.0	11-10-2025	Adaptación de los materiales a markdown
1.1	06-11-2025	Ampliación con sección ver funciones y procedimientos almacenados

Introducción

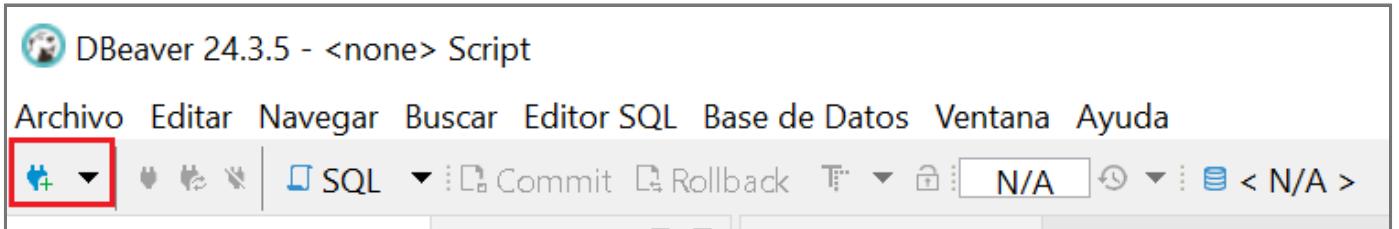
DBeaver es una herramienta gráfica y gratuita que permite gestionar múltiples bases de datos de forma visual. Algunas de las acciones que podemos realizar con esta herramienta son las siguientes:

- Explorar la estructura de la base de datos (tablas, vistas, claves, relaciones...).
- Consultar datos.
- Modificar tablas, añadir registros o ejecutar scripts SQL sin salir del proyecto.
- Probar consultas antes de implementarlas en el programa.

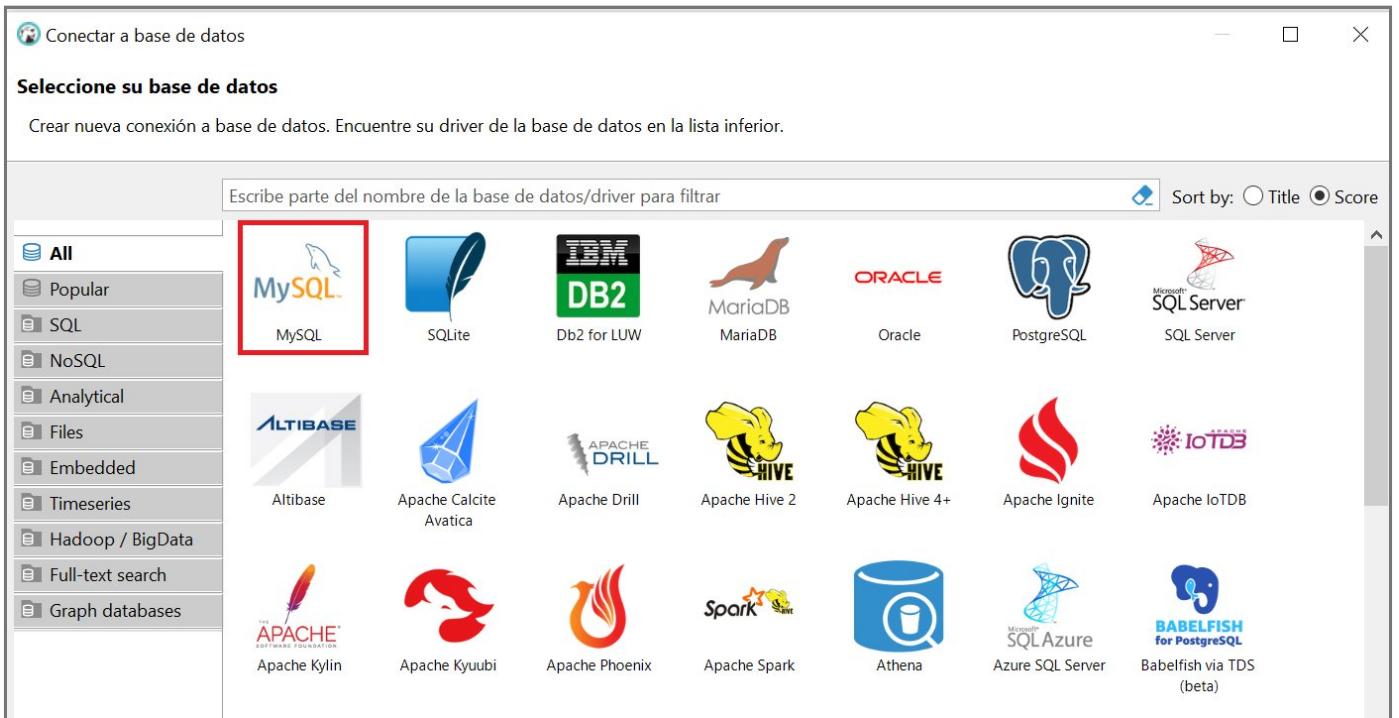
Conexión a MySQL

Para conectar a una base de datos *MySQL* sigue estos pasos:

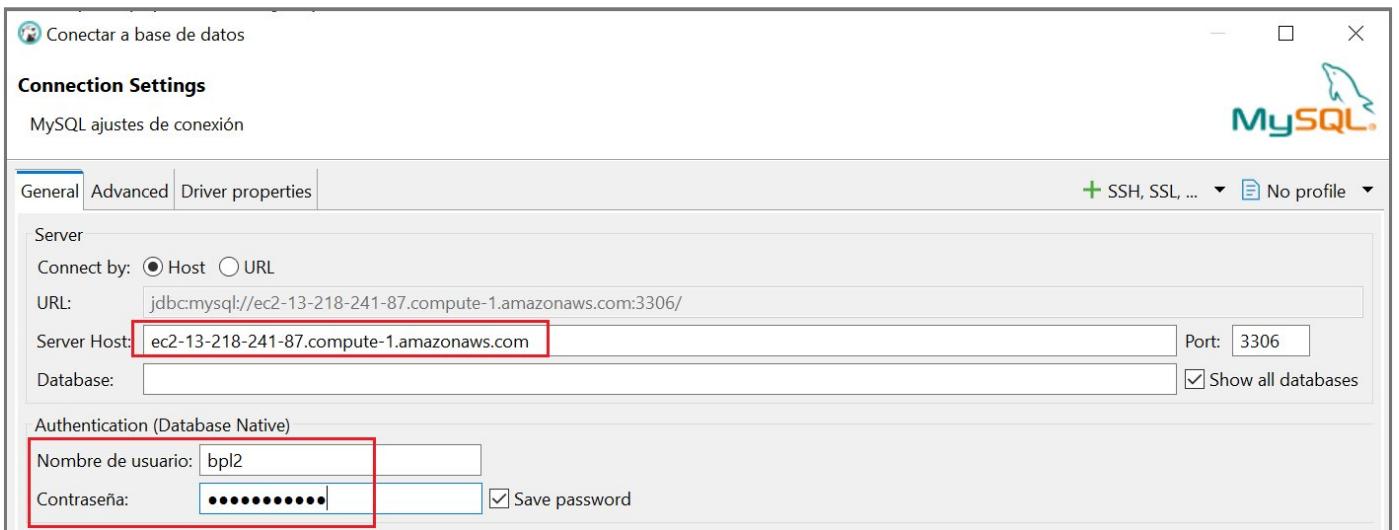
1. Haz clic en el botón Nueva conexión (ícono de enchufe) o entra al menú Archivo > Nueva conexión



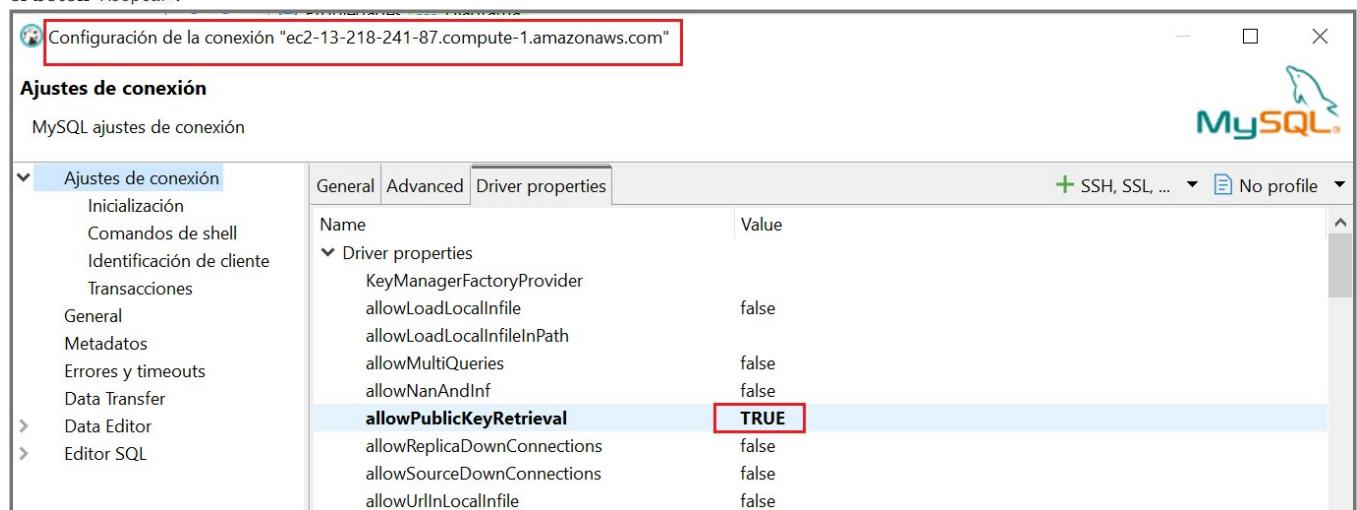
2. Selecciona MySQL y pulsa en el botón Siguiente



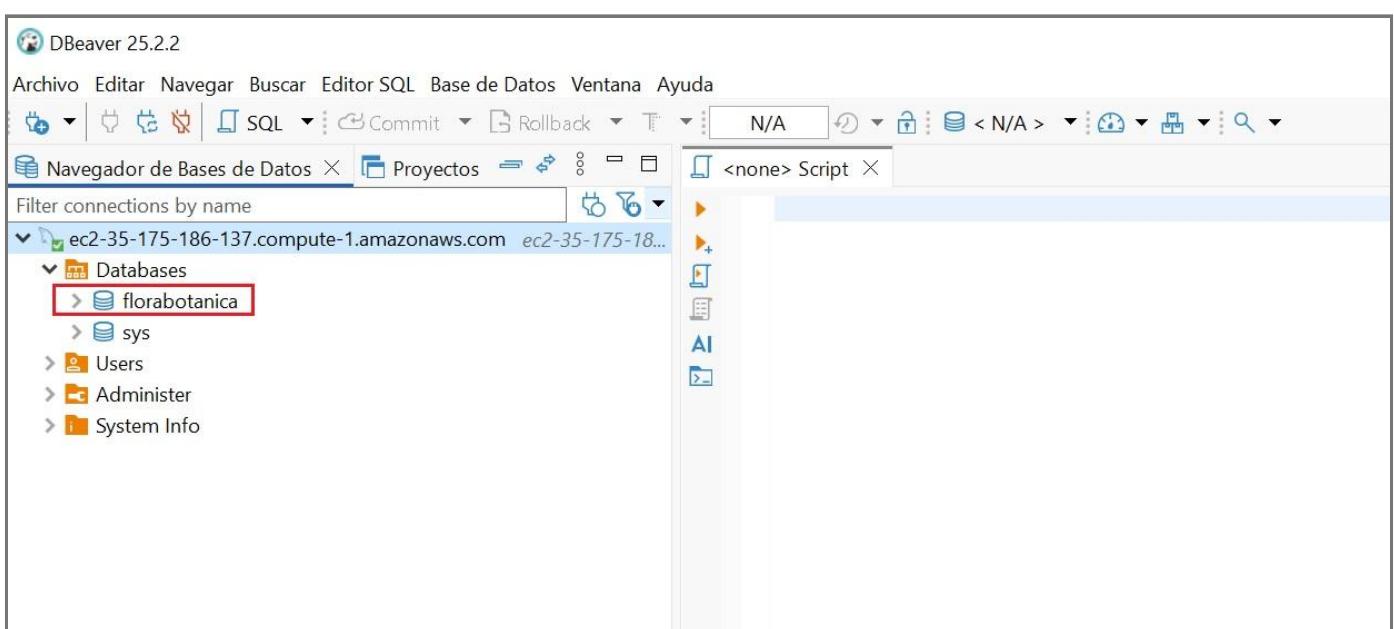
3. Indica los datos del servidor, usuario y contraseña. Si quieres ver todas las bases de datos a las que el usuario puede acceder deja marcada la casilla Show all database y no indiques nada en la casilla database



Si aparece Error "Public Key Retrieval is not allowed" haz clic con el botón derecho en tu conexión y selecciona **Editar conexión** luego ve a la pestaña **Driver Properties** y cambia la propiedad **allowPublicKeyRetrieval** a **TRUE** (por defecto está a **false**). Luego haz clic en el botón **Aceptar**.



4. Una vez conectado, verás las bases de datos del servidor



Ver funciones y procedimientos almacenados

Para poder ver el código de una función o un procedimiento almacenado en nuestra BD seguir estos pasos:

1. Desplegar el apartado Procedures de la BD.

2. Hacer clic con el botón derecho sobre la función o procedimiento, entrar en Generar SQL y luego en DDL.

DBeaver 25.2.2 - jardines

Archivo Editar Navegar Buscar Editor SQL Base de Datos Ventana Ayuda

Navegador de Bases de Datos Proyectos

Filter connections by name

> database-aurora-bpaterna.cluster-cfxl6jsrig9d.us-east-1.amazonaws.com database-aurora-bpaterna...

✓ florabotanica mysql EC2 100.25.102.165:3306

Databases

- florabotanica
 - Tables
 - jardines
 - jardines_plantas
 - plantas
 - Views
 - Indexes
 - Procedures
 - fn_total_valor_pl
 - sp_agregar_plan
 - sp_listar_plantas
 - Triggers
 - Events
- sys

Files - General

Name

Bookmarks

Generar SQL

CALL

DDL

Alt+Insertar F4

Ver Procedure

Filter

Compare/Migrate

Herramientas

Copiar Ctrl+C

1	1	Apolo
2	2	Atenea
3	3	Iris

El código de la función o procedimiento aparecerá en una ventana nueva.

Generated SQL (florabotanica mysql EC2)

Previsualización de SQL:

```
CREATE DEFINER=`bp13`@`%` FUNCTION `florabotanica`.`fn_total_valor_planta`(p_id_planta INT) RETURNS double
  DETERMINISTIC
BEGIN
  DECLARE total DOUBLE;

  SET total = (
    SELECT stock * precio
    FROM plantas
    WHERE id_planta = p_id_planta);

  RETURN total;

END;
```

Settings

Usar nombres completos Compartir SQL

Refresh Copiar Cerrar

Autoría

Obra realizada por Begoña Paterna Lluch. Publicada bajo licencia [Creative Commons Atribución/Reconocimiento-CompartirIgual 4.0 Internacional](#)

3.4 MySQL



Revisiones

Revisión	Fecha	Descripción
1.0	10-11-2025	Adaptación de los materiales a markdown
1.0	15-01-2026	Instalación en Docker

3.4.1 1. Introducción

MySQL es un sistema de gestión de bases de datos relacional (RDBMS), basado en el lenguaje SQL (Structured Query Language). Fue desarrollado originalmente por MySQL AB, luego adquirido por Sun Microsystems y actualmente es propiedad de Oracle Corporation. Se utiliza ampliamente en aplicaciones web y empresariales para almacenar, organizar y acceder a datos de manera eficiente.

MySQL funciona bajo una arquitectura cliente-servidor, en la cual el servidor MySQL es el componente encargado de almacenar, gestionar y proteger las bases de datos, mientras que los clientes son las aplicaciones o usuarios que se conectan a él para consultar o manipular los datos mediante el lenguaje SQL.

En estos apuntes vamos a utilizar como servidor una instancia ECS de AWS, puedes ver como instalarlo desde [AWS Learner Lab](#). Una vez tengas tu servidor funcionando, ya puedes instalar el servidor MySQL en él.

3.4.2 2. Instalación en EC2

1. Conectar al servidor por ssh

Para conectar, abre una ventana de comandos y asegurate que el archivo .pem está en la carpeta desde la que lanzas el siguiente comando (puedes utilizar el nombre del servidor o su IP pública):

```
ssh -i [nombre_clave] ubuntu@[nombre_IP_servidor]
```

Si aparece el siguiente aviso:

```
@@@@@@@
@      WARNING: UNPROTECTED PRIVATE KEY FILE!      @
@@@@@@@
Permissions 0644 for '[REDACTED]' are too open.
It is required that your private key files are NOT accessible by others.
This private key will be ignored.
Load key "[REDACTED]": bad permissions
[REDACTED]: Permission denied (publickey).
```

Ejecuta el comando siguiente:

```
chmod 400 [nombre_clave]
```

2. Actualiza la lista de paquetes del servidor

```
sudo apt update
```

3. Instala el servidor MySQL y las dependencias necesarias

```
sudo apt install mysql-server
```

4. Comprueba el estado del servicio

Comprueba que el servicio de MySQL se esté ejecutando correctamente (Si no está activo, puedes iniciararlo con `sudo systemctl start mysql`)

```
sudo systemctl status mysql
```

```
ubuntu@ip-172-31-16-226:~$ sudo systemctl status mysql
● mysql.service - MySQL Community Server
  Loaded: loaded (/usr/lib/systemd/system/mysql.service; enabled; preset: enabled)
  Active: active (running) since Sat 2025-10-11 15:06:25 UTC; 4min 38s ago
    Process: 2396 ExecStartPre=/usr/share/mysql/mysql-systemd-start pre (code=exited, status=0/SUCCESS)
   Main PID: 2405 (mysql)
     Status: "Server is operational"
       Tasks: 39 (limit: 1008)
      Memory: 353.1M (peak: 377.0M)
        CPU: 3.117s
      CGroup: /system.slice/mysql.service
              └─2405 /usr/sbin/mysqld

Oct 11 15:06:24 ip-172-31-16-226 systemd[1]: Starting mysql.service - MySQL Community Server...
Oct 11 15:06:25 ip-172-31-16-226 systemd[1]: Started mysql.service - MySQL Community Server.
```

Crea un usuario y una base de datos

1. Entra al servidor

Entra al servidor MySQL (cuando te pida contraseña déjala en blanco y pulsa `INTRO`)

```
sudo mysql -u root -p
```

2. Crea el usuario con su contraseña

Ejecuta los comandos siguientes (el % indica que el usuario podrá conectarse desde cualquier sitio). Cambia [tu_usuario] y [tu_contraseña] por el usuario y contraseña que quieras:

```
CREATE USER '[tu_usuario]@'%' IDENTIFIED BY '[tu_contraseña]';
GRANT ALL PRIVILEGES ON *.* TO '[tu_usuario]@'%';
FLUSH PRIVILEGES;
SHOW GRANTS FOR '[tu_usuario]@'%';
```

3. Crea la base de datos

Por ejemplo (cambia el nombre del ejemplo por el de tu BD)

```
create database [nombre_BD];
```

4. Sal del servidor

```
exit
```

Configura MySQL y el servidor para permitir conexiones externas

1. Edita el fichero de configuración

```
sudo nano /etc/mysql/mysql.conf.d/mysqld.cnf
```

Busca la instrucción siguiente:

```
bind-address = 127.0.0.1
```

Modifícalo para que quede así:

```
# bind-address = 127.0.0.1
bind-address = 0.0.0.0
```

Guarda los cambios, reinicia el servicio y comprueba que ha arrancado correctamente

```
sudo systemctl restart mysql
sudo systemctl status mysql
```

2. Configura el servidor para permitir tráfico entrante

Añade una regla en el servidor para permitir el tráfico entrante del puerto 3306. Para ello haz clic en la pestaña Seguridad y luego en el enlace de Grupos de seguridad

The screenshot shows the AWS EC2 Instances page. On the left sidebar, under 'Instancias', 'Instancias' is selected. In the main area, there is one instance listed: 'i-0193981641dd0ccbfb (bpl)'. Below the instance card, the 'Seguridad' tab is highlighted with a red box. Other tabs include 'Detalles', 'Estado y alarmas', 'Monitoreo', 'Redes', 'Almacenamiento', and 'Etiquetas'.

Entra en Reglas de entrada y haz clic en el botón Editar reglas de entrada

The screenshot shows the AWS Security Groups page. Under 'Instancias', 'Grupos de seguridad' is selected. A specific security group, 'sg-02aa0093d9706111a - launch-wizard-1', is shown. The 'Reglas de entrada' tab is selected. At the bottom right of this section, the 'Editar reglas de entrada' button is highlighted with a red box.

Haz clic en Agregar regla, configura el tipo, el puerto y la IP de origen 0.0.0.0/0 para permitir acceso desde cualquier lugar y por último haz clic en el botón Guardar reglas

En unos segundos aparecerá tu nueva regla en la lista

ID de la regla del grupo de seguridad	Tipo	Protocolo	Intervalo de puertos	Origen	Descripción: opcional
sgr-0d0be3626cd926b6d	SSH	TCP	22	Pers...	
-	TCP personalizado	TCP	3306	Any...	0.0.0.0/0 X 0.0.0.0/0 X

3. Prueba de conexión

Prueba a conectar a tu base de datos desde DBeaver

Exportación de la BD

1. Conecta al servidor por ssh

Para conectar, abre una ventana de comandos y asegurate que el archivo .pem está en la carpeta desde la que lanzas el siguiente comando (puedes utilizar el nombre del servidor o su IP pública):

```
ssh -i [nombre_clave] ubuntu@[nombre_IP_servidor]
```

2. Crea un archivo con la exportación

Para hacer un dump de la BD ejecuta ():

```
mysqldump -u [tu_usuario] -p --routines [tu_BD] > [nombre_archivo_dump].sql
```

Después comprueba que el archivo se ha creado y cierra sesión con el comando `exit`

3. Descarga el archivo

Para descargar al equipo local utiliza el comando (luego comprueba que el archivo se ha descargado correctamente y que su contenido es correcto):

```
scp -i [nombre_certificado] ubuntu@[IP_nombre_servidor]:[ruta_archivo_dump].sql [ruta_destino]
```

3.4.3 3. Instalación en Docker

Partimos de un sistema operativo con Docker ya funcionando y listo para poder crear contenedores.

Algunos comandos útiles para manejarnos en docker

- Comprobar si Docker está instalado en nuestro sistema operativo

```
docker --version
```

- Ver los contenedores creados (incluidos los detenidos)

```
docker ps -a
```

- Iniciar un contenedor

```
docker start <nombre_o_id_del_contenedor>
```

- Detener un contenedor

```
docker stop <nombre_o_id_del_contenedor>
```

- Eliminar un contenedor (detenerlo antes)

```
docker rm <nombre_o_id_del_contenedor>
```

Crear el contenedor MySQL

A continuación se describen los pasos para crear un contenedor que hará de servidor MySQL. En la configuración se establece la contraseña del usuario `root` para el servidor y se crea una base de datos llamada `florabotanica` con un usuario y contraseña para acceder a ella.

Crear archivo llamado `docker-compose.yml` con el contenido siguiente:

```
version: '3.8'

services:
  mysql:
    image: mysql:8.0
    container_name: mysql-server
    ports:
      - "3306:3306"
    environment:
      MYSQL_ROOT_PASSWORD: <password>
      MYSQL_DATABASE: florabotanica
      MYSQL_USER: flora
      MYSQL_PASSWORD: <password>
```

Desde una ventana de terminal o utilizando la línea de comandos ejecutar:

```
docker compose up -d
```

Crear tabla una tabla en la base de datos

Para entrar al contenedor ejecutar:

```
docker exec -it mysql-server bash
```

Conectarse a la base de datos `florabotanica` con el usuario `flora` (pedirá contraseña) con el comando:

```
mysql -u flora -p florabotanica
```

Crear una tabla llamada `plantas` ejecutando las líneas:

```
CREATE TABLE plantas (
    id_planta INT AUTO_INCREMENT PRIMARY KEY,
    nombre TEXT NOT NULL,
    tipo TEXT NOT NULL,
    foto TEXT,
    altura DOUBLE) ENGINE=InnoDB;
```

Comprobar que se ha creado correctamente con el comando:

```
SHOW TABLES;
```

Realizar alguna inserción, por ejemplo:

```
INSERT INTO plantas (nombre, tipo, altura) VALUES ('Rosa', 'Flor', 1.2);
```

Comprobar que la tabla contiene los datos con el comando:

```
SELECT * FROM plantas;
```

Exportación de nuestra base de datos

El siguiente comando crea un archivo llamado `florabotanica.sql` con la estructura y datos de la base de datos llamada `florabotanica` que está alojada en un servidor MySQL en un contenedor Docker llamado `mysql-server`. Al ejecutar el comando (desde terminal o línea de comandos) el sistema pedirá la contraseña de root y el archivo se creará en la carpeta desde donde nos encontramos):

```
docker exec -i mysql-server mysqldump --no-tablespaces -u root -p florabotanica > florabotanica.sql
```

Importación de nuestra base de datos (opción 1)

Para importar en otro contenedor con un servidor MySQL la estructura y datos de la base de datos llamada `florabotanica`, ejecutar desde terminal o línea de comandos en la carpeta donde está ubicado el archivo `florabotanica.sql` el siguiente comando (pedirá la contraseña de root). Se asume que el contenedor se llama `mysql-server2`:

```
docker exec -i mysql-server2 mysql -u root -p florabotanica < florabotanica.sql
```

Importación de nuestra base de datos (opción 2)

Si la opción anterior da problemas seguir estos pasos (se asumen los mismos nombres de contenedor, base de datos y fichero que en la opción anterior):

Abrir terminal o cmd y copiar el archivo .sql al contenedor

```
docker cp florabotanica.sql mysql-server2:/tmp/florabotanica.sql
```

Abrir shell Bash dentro del contenedor:

```
docker exec -it mysql-server bash
```

Entrar a mysql como root

```
mysql -p
```

Crear BD y restaurar

```
mysql> CREATE DATABASE florabotanica;
mysql> USE florabotanica;
mysql> SOURCE /tmp/florabotanica.sql;
```

Autoría

Obra realizada por Begoña Paterna Lluch. Publicada bajo licencia [Creative Commons Atribución/Reconocimiento-CompartirIgual 4.0 Internacional](#)

3.5 Instalación y administración de MongoDB

Revisões

Revisión	Fecha	Descripción
1.0	10-11-2025	Adaptación de los materiales a markdown
1.1	15-11-2025	Sección MongoDB en EC2 (AWS)
1.2	15-12-2025	Servidor en Docker

Opciones de instalación y despliegue

Opción	Descripción	Ideal para
MongoDB Community Server	Versión gratuita que se instala localmente en Windows, Linux o macOS.	Prácticas locales, entornos educativos.
MongoDB en Docker	Se ejecuta como contenedor con <code>docker-compose</code> o comandos <code>docker run</code> .	Entornos de desarrollo rápidos y reproducibles.
MongoDB Atlas	Servicio en la nube oficial de MongoDB. Permite crear clústeres gratuitos o de pago, gestionados por Mongo.	Proyectos web, microservicios, despliegues reales.
MongoDB Local + Atlas Sync	Permite sincronizar datos locales con una base remota en Atlas.	Aplicaciones con modo offline/online.

Herramientas de administración y visualización

Herramienta	Tipo	Descripción
MongoDB Compass	GUI oficial	Interfaz gráfica para consultar, insertar y analizar datos.
DBeaver	GUI universal	Permite conectarse a Mongo y a otras bases de datos (SQL y NoSQL).
Robo 3T (antiguo Robomongo)	GUI ligera	Muy utilizada para tareas básicas de exploración.
mongosh	Consola oficial	Shell de comandos moderno (sustituye a <code>mongo</code>).

De entre todas las opciones posibles para instalar y administrar MongoDB, utilizaremos la versión **Community** junto con **Mongo Shell (mongosh)** por su simplicidad, ligereza y adecuación a los objetivos de esta unidad.

Instalación del servidor (Linux)

Descargamos la versión apropiada para el sistema operativo con el que se está trabajando desde la página oficial de MongoDB: <https://www.mongodb.com/try/download/community>. Para ello entramos al menú **Products → Community Edition → Community Server**. (Para la realización de estos apuntes se ha descargado la versión 8.2.1 para Ubuntu 22.04 x64 en formato .tgz)

Descomprimimos el archivo descargado (para facilitar el trabajo se puede renombrar la carpeta descomprimida a `mongodb`). Después, dentro de la carpeta `mongodb`, creamos un directorio llamado `data` y dentro de él otro llamado `db`. Por último arrancamos el servidor ejecutando en una ventana de terminal el comando:

```
mongodb/bin/mongod --dbpath mongodb/data/db
```

Si el servidor ha arrancado correctamente, aparecerán una serie de mensajes informativos y el servidor quedará en espera de recibir peticiones del cliente:

```
b.paternalluch@lliurex-desktop:~$ mongoDB/bin/mongod --dbpath mongoDB/data/db
{"t":{"$date":"2025-11-04T18:51:07.999+01:00"},"s":"I", "c": "-", "id":8991200, "ctx":"main","msg":"Shuffling initializers","attr":{"seed":2973834231}}
{"t":{"$date":"2025-11-04T18:51:08.022+01:00"},"s":"I", "c":"CONTROL", "id":97374, "ctx":"main","msg":"Automatically disabling TLS 1.0 and TLS 1.1, to force-enable TLS 1.1 specify --sslDisabledProtocols 'TLS1_0'; to force-enable TLS 1.0 specify --sslDisabledProtocols 'none'"}
{"t":{"$date":"2025-11-04T18:51:08.026+01:00"},"s":"I", "c":"NETWORK", "id":4915701, "ctx":"main","msg":"Initialized wire specification","attr":{"spec":{"incomingExternalClient":{"minWireVersion":0,"maxWireVersion":27}, "incomingInternalClient":{"minWireVersion":0,"maxWireVersion":27}, "outgoing":{"minWireVersion":6,"maxWireVersion":27}, "isInternalClient":true}}}
{"t":{"$date":"2025-11-04T18:51:08.027+01:00"},"s":"I", "c":"CONTROL", "id":5945603, "ctx":"main","msg":"Multi threading initialized"}
{"t":{"$date":"2025-11-04T18:51:08.027+01:00"},"s":"I", "c":"CONTROL", "id":4615611, "ctx":"initandlisten", "attr":{"connectionId":1, "client": "MongoDB shell version: 5.0.20", "version": "5.0.20", "type": "Electron"}, "msg":"Received first command on ingress connection since session start or auth handshake","attr":{"elapsedMillis":1}}
{"t":{"$date":"2025-11-04T17:55:54.640+01:00"},"s":"I", "c":"NETWORK", "id":6788700, "ctx":"conn4","msg":"Connection accepted","attr":{"remote":"127.0.0.1:52432","isLoadBalanced":false,"uuid":{"$uuid":57179bb6-4986-4fd0-8ac3-3eb3d8a8d064}},"connectionId":5,"connectionCount":5}
{"t":{"$date":"2025-11-04T17:56:05.131+01:00"},"s":"I", "c":"NETWORK", "id":22943, "ctx":"listener","msg":"Connection accepted","attr":{"remote":"127.0.0.1:52432","isLoadBalanced":false,"uuid":{"$uuid":57179bb6-4986-4fd0-8ac3-3eb3d8a8d064}},"connectionId":5,"connectionCount":5}
{"t":{"$date":"2025-11-04T17:56:05.132+01:00"},"s":"I", "c":"NETWORK", "id":51800, "ctx":"conn5","msg":"Connection not authenticating","attr":{"client":"127.0.0.1:52432","doc":{"application":{"name":"mongosh 2.5.9"}, "driver":{"name":"nodejs|mongosh", "version":"6.19.0|2.5.9"}, "platform":"Node.js v20.19.5, LE", "os":{"name":"linux", "architecture":"x64", "version":"3.10.0-327.22.2.el7.x86_64", "type":"Linux"}}, "connectionId":5,"connectionCount":5}
{"t":{"$date":"2025-11-04T17:56:05.132+01:00"},"s":"I", "c":"ACCESS", "id":10483900, "ctx":"conn5","msg":"Connection not authenticating","attr":{"client":"127.0.0.1:52432","doc":{"application":{"name":"mongosh 2.5.9"}, "driver":{"name":"nodejs|mongosh", "version":"6.19.0|2.5.9"}, "platform":"Node.js v20.19.5, LE", "os":{"name":"linux", "architecture":"x64", "version":"3.10.0-327.22.2.el7.x86_64", "type":"Linux"}}, "connectionId":5,"connectionCount":5}
```

No se debe cerrar esa ventana de terminal, porque el servidor se detendrá.

Alternativa: Contenedor docker

Como alternativa al servidor en los ordenadores de clase podemos crear un contenedor con el servidor de MongoDB siguiendo estos pasos (los ordenadores de clase ya tienen instalado Docker):

Podemos crearlo de dos formas:

1. Para que se inicie cuando arranque el sistema: `docker run --name [nombre_contenedor] -d -p 27017:27017 --restart unless-stopped mongo:4.4`
2. Para que no se inicie cuando arranque el sistema (habrá que iniciarla manualmente): `docker run --name [nombre_contenedor] -d -p 27017:27017 mongo:4.4`

Para ver los contenedores (incluyendo los detenidos): `docker ps -a`

Para iniciar un contenedor: `docker start [nombre_contenedor]`

Instalación del cliente Mongo Shell (Linux)

Descargamos la versión apropiada para el sistema operativo con el que se está trabajando desde la página oficial de MongoDB: <https://www.mongodb.com/try/download/shell> Para ello entramos al menú **Products → Tools → MongoDB Shell**. (Para la realización de estos apuntes se ha descargado la versión 2.5.9 para Linux 64 en formato .tgz)

Descomprimimos el archivo descargado (para facilitar el trabajo se ha renombrado la carpeta descomprimida a `mongosh`) y arrancamos el cliente ejecutando en una nueva ventana de terminal el comando siguiente:

```
mongosh/bin/mongosh
```

Aparecerá la siguiente información:

```
b.paternalluch : mongod x b.paternalluch : mongosh mongodb x
b.paternalluch@lliurex-desktop:~$ mongosh/bin/mongosh
Current Mongosh Log ID: 690a3d3ab8334e5ec19dc29c
Connecting to:      mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.5.9
Using MongoDB:     8.2.1
Using Mongosh:    2.5.9

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

To help improve our products, anonymous usage data is collected and sent to MongoDB periodically (https://www.mongodb.com/legal/privacy-policy).
You can opt-out by running the disableTelemetry() command.

-----
The server generated these startup warnings when booting
2025-11-04T18:51:08.029+01:00: Using the XFS filesystem is strongly recommended with the WiredTiger storage engine. See http://dochub.mongodb.org/core/prodnotes-filesystem
2025-11-04T18:51:08.442+01:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
2025-11-04T18:51:08.442+01:00: This server is bound to localhost. Remote systems will be unable to connect to this server. Start the server with --bind_ip <address> to specify which IP addresses it should serve responses from, or with --bind_ip_all to bind to all interfaces. If this behavior is desired, start the server with --bind_ip 127.0.0.1 to disable this warning
2025-11-04T18:51:08.442+01:00: Soft rlimits for open file descriptors too low
2025-11-04T18:51:08.442+01:00: For customers running the current memory allocator, we suggest changing the contents of the following sysfsFile
2025-11-04T18:51:08.443+01:00: For customers running the current memory allocator, we suggest changing the contents of the following sysfsFile
2025-11-04T18:51:08.443+01:00: We suggest setting the contents of sysfsFile to 0.
2025-11-04T18:51:08.443+01:00: Your system has glibc support for rseq built in, which is not yet supported by tcmalloc-google and has critical performance implications. Please set the environment variable GLIBC_TUNABLES=glibc.pthread.rseq=0
2025-11-04T18:51:08.443+01:00: We suggest setting swappiness to 0 or 1, as swapping can cause performance problems.
-----
test> 
```

Para comprobar el funcionamiento ejecutamos el siguiente comando:

```
show dbs
```

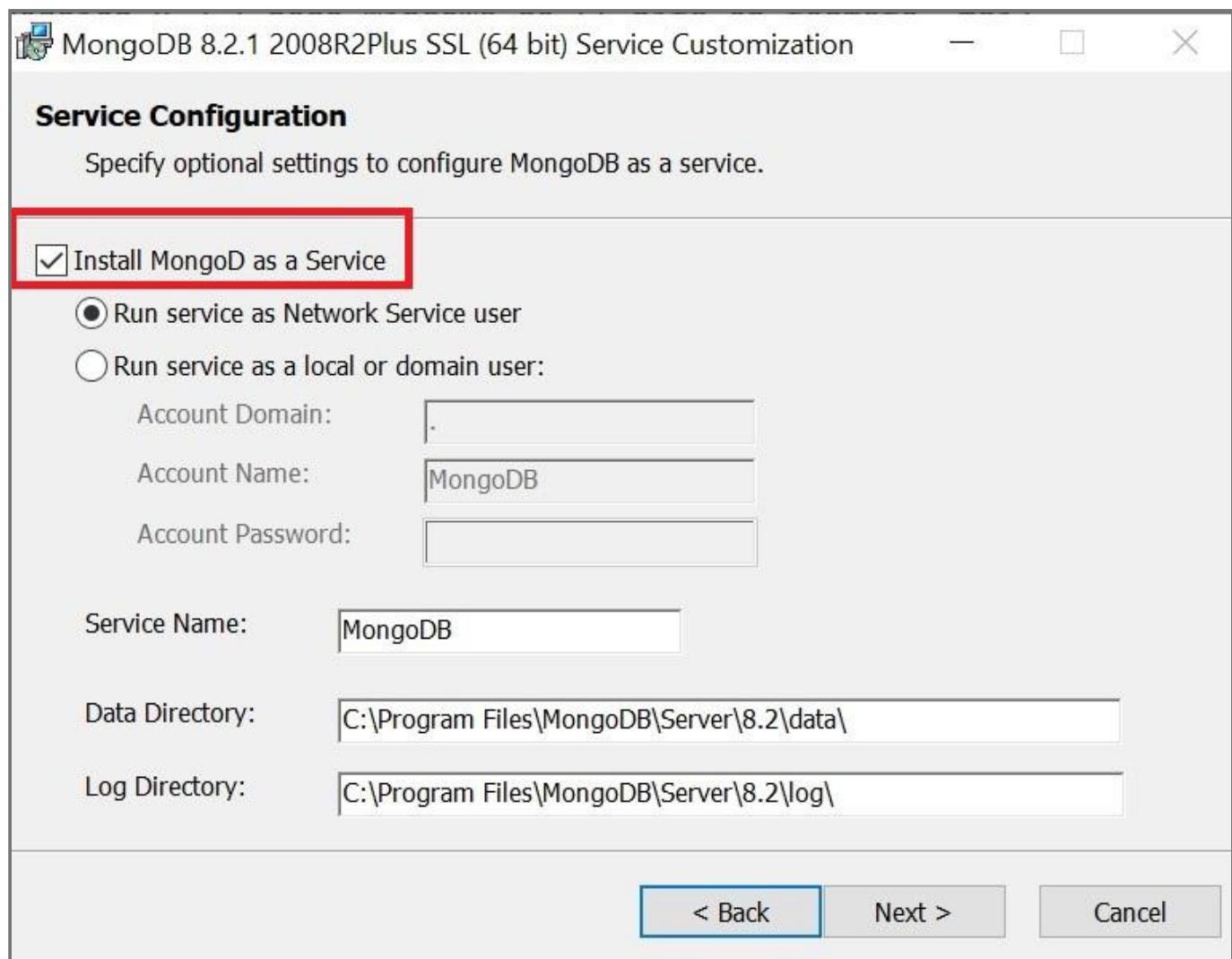
Si aparecen las bases de datos (admin, config, local), todo está funcionando correctamente, son las bases de datos del sistema. Podemos ver que al final de la pantalla aparece la palabra `test>` es porque en realidad, estamos conectados a una base de datos llamada `test`.

```
test> show dbs
admin   40.00 KiB
config  12.00 KiB
local   72.00 KiB
test> 
```

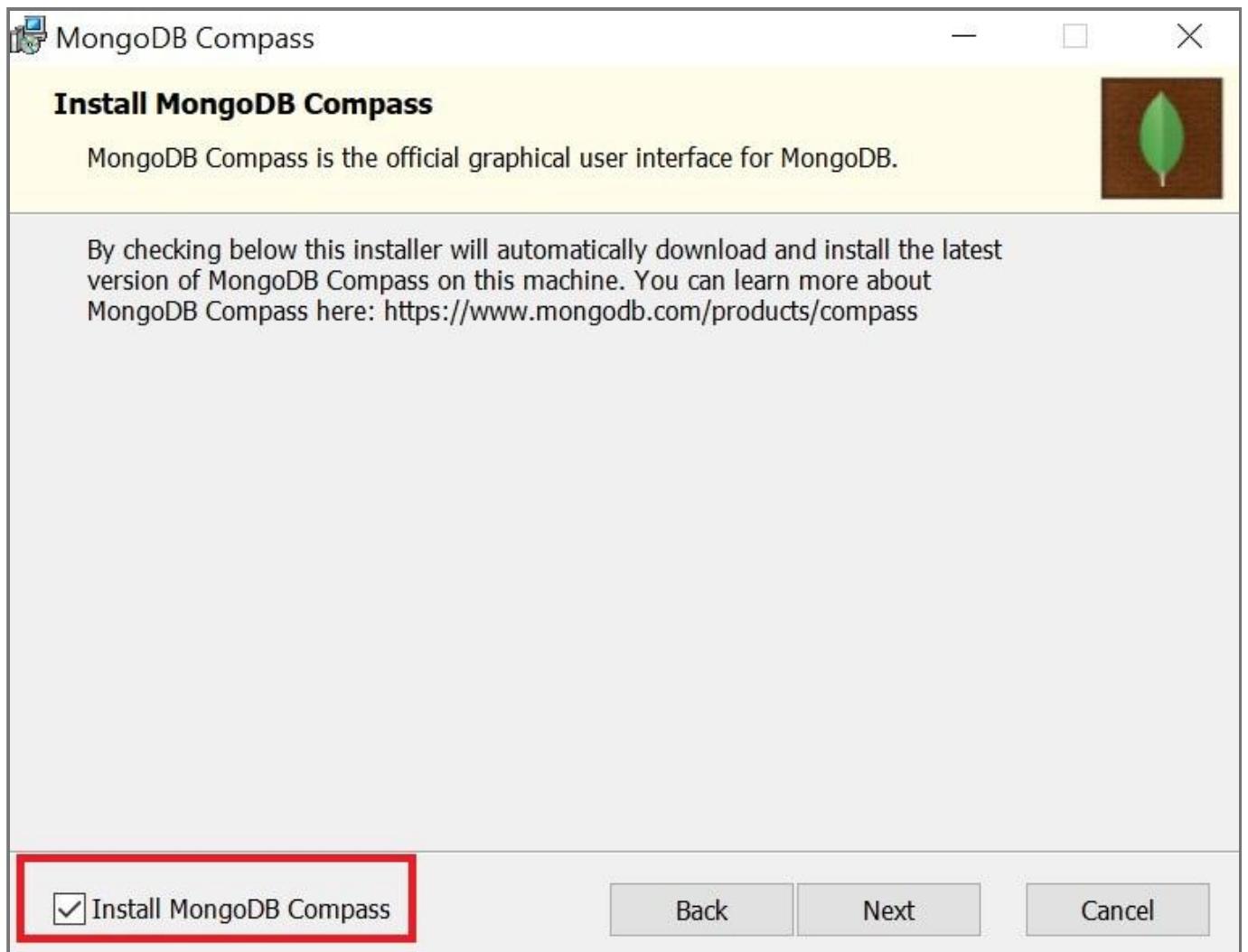
Instalación del servidor (Windows)

Descargamos la versión apropiada para el sistema operativo con el que se está trabajando desde la página oficial de MongoDB: <https://www.mongodb.com/try/download/community> Para ello entramos al menú **Products → Community Edition → Community Server**. (Para la realización de estos apuntes se ha descargado la versión 8.2.1 para Windows de 64 bits en formato .msi)

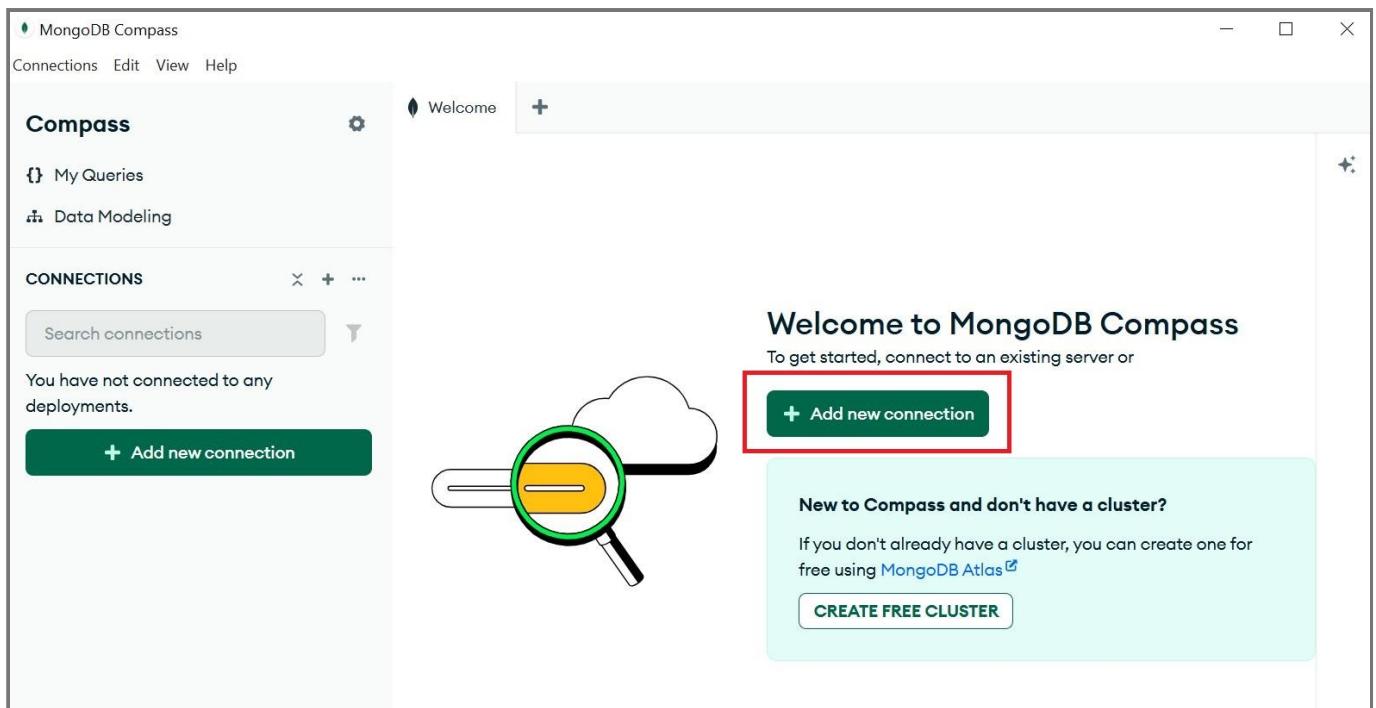
Durante la instalación marcamos la opción de instalar MongoDB como servicio para que el programa se iniciará automáticamente con el sistema.



Durante la instalación también podemos marcar la opción de instalar **MongoDB Compass**, que es la herramienta gráfica oficial de MongoDB, la cual permite visualizar, explorar y administrar bases de datos MongoDB sin necesidad de utilizar la línea de comandos.



Una vez finalizada la instalación podemos arrancar el cliente y ver que nos pide que creamos una nueva conexión:



New Connection

Manage your connection settings

URI [i](#) **Edit Connection String**

mongodb://localhost:27017/

Name **Color** [▼](#)

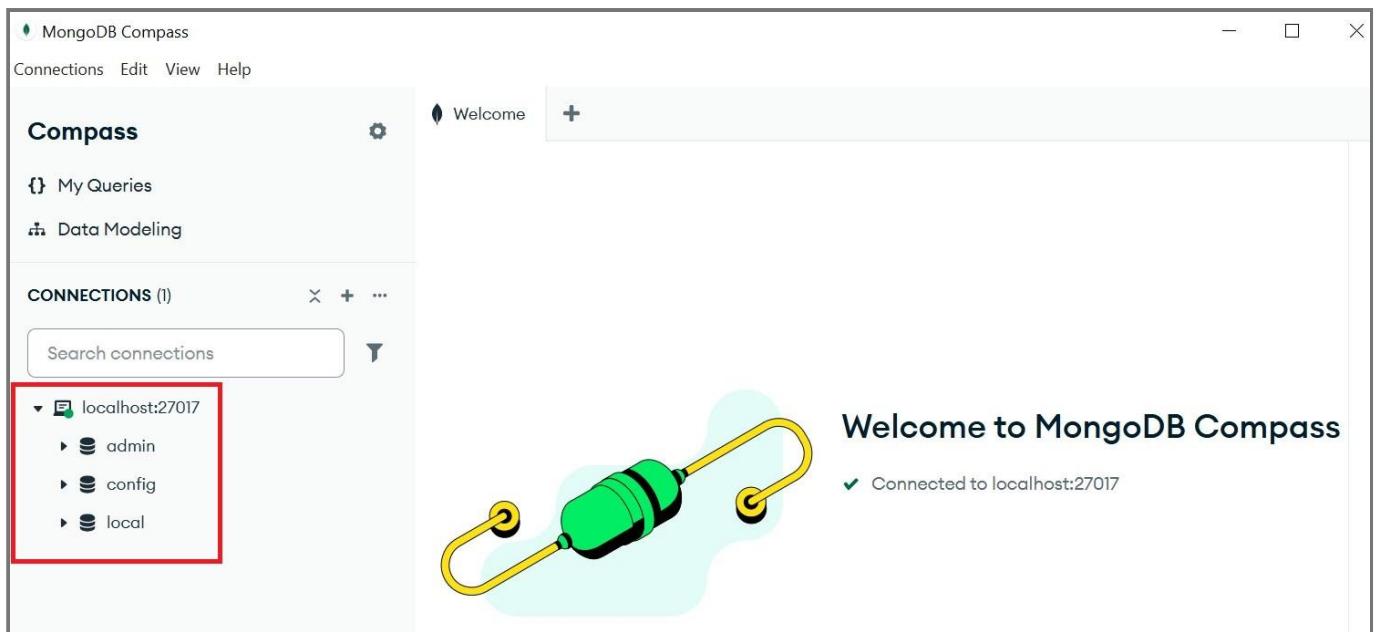
Cancel **Save** **Connect** **Save & Connect**

How do I find my connection string in Atlas?

If you have an Atlas cluster, go to the Cluster view. Click the 'Connect' button for the cluster to which you wish to connect.

[See example](#)

Una vez conectados (no hace falta indicar ningún dato de conexión), veremos las bases de datos del sistema:



Si preferimos comunicarnos con el servidor por comandos, podemos instalar el cliente Mongo Shell.

Instalación del cliente Mongo Shell (Windows)

Podemos descargarlo desde la página oficial: <https://www.mongodb.com/try/download/shell>. Para ello entramos al menú **Products → Tools → MongoDB Shell**. (Para la realización de estos apuntes se ha descargado la versión 2.5.9 para Windows de 64 bits en formato .msi)

Escribe el siguiente comando en una ventana de terminal:

```
mongosh
```

Si el servidor está arrancado aparecerá la siguiente información:

```
mongosh mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000
Microsoft Windows [Versión 10.0.19045.6456]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\bego>mongosh
Current Mongosh Log ID: 690b11b82e2763527463b111
Connecting to:      mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.5.9
Using MongoDB:      8.2.1
Using Mongosh:      2.5.9

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

To help improve our products, anonymous usage data is collected and sent to MongoDB periodically (https://www.mongodb.com/legal/privacy-policy).
You can opt-out by running the disableTelemetry() command.

-----
The server generated these startup warnings when booting
2025-11-04T22:49:49.740+01:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
-----
test>
```

Para comprobar el funcionamiento ejecuta el siguiente comando:

```
show dbs
```

Si aparecen las bases de datos (admin, config, local), todo está funcionando correctamente, son las bases de datos del sistema. Podemos ver que al final de la pantalla aparece la palabra `test>` es porque en realidad, estamos conectados a una base de datos llamada test.

```
test> show dbs
admin      40.00 KiB
config     108.00 KiB
local      40.00 KiB
test> -
```

Instalación de MongoDB en EC2 (AWS)

A continuación se describen los pasos para instalar y configurar MongoDB en nuestra instancia EC2 de AWS.

Conectarse al servidor por ssh

Para conectar, abre una ventana de comandos y asegúrate que el archivo `.pem` está en la carpeta desde la que lanzas el siguiente comando (puedes utilizar el nombre del servidor o su IP pública). Sustituye `[nombre_clave]` por el nombre del archivo de tu clave y `[nombre_IP_servidor]` por el nombre o IP de tu servidor:

```
ssh -i [nombre_clave] ubuntu@[nombre_IP_servidor]
```

Si aparece el siguiente aviso:

```
@@@@@@@
@      WARNING: UNPROTECTED PRIVATE KEY FILE!      @
@@@@@@@
Permissions 0644 for '[REDACTED]' are too open.
It is required that your private key files are NOT accessible by others.
This private key will be ignored.
Load key "[REDACTED]": bad permissions
[REDACTED]: Permission denied (publickey).
```

Ejecuta el comando siguiente:

```
chmod 400 [nombre_clave]
```

Instalar MongoDB Community Edition

1. Importar la clave pública

```
sudo apt install -y curl gnupg
curl -fsSL https://pgp.mongodb.com/server-7.0.asc | \
sudo gpg -o /usr/share/keyrings/mongodb-server-7.0.gpg --dearmor
```

2. Crear una lista de verificación

```
echo "deb [signed-by=/usr/share/keyrings/mongodb-server-7.0.gpg] https://repo.mongodb.org/apt/ubuntu jammy/mongodb-org/7.0 multiverse" | \
sudo tee /etc/apt/sources.list.d/mongodb-org-7.0.list
```

3. Actualizar lista de paquetes

```
sudo apt update
```

4. Instalar MongoDB Community Server

```
sudo apt install -y mongodb-org
```

Iniciar y verificar el funcionamiento

1. Iniciar el servidor

```
sudo systemctl start mongod
```

Si aparece un error parecido a Failed to start mongod.service: Unit mongod.service not found, ejecutar estos dos comandos:

```
sudo systemctl daemon-reload
sudo systemctl start mongod
```

2. Comprobar el estado del servidor

```
sudo systemctl status mongod
```

3. Configurar para que se inicie al arrancar el sistema

```
sudo systemctl enable mongod
```

4. Parar y reiniciar

```
sudo systemctl stop mongod
sudo systemctl restart mongod
```

Configurar acceso remoto

1. Editar el fichero de configuración

```
sudo nano /etc/mongod.conf
```

Busca el bloque de instrucciones siguiente:

```
# network interfaces
net:
port: 27017
bindIp: 127.0.0.1
```

Modifícalo para que quede así:

```
# network interfaces
net:
port: 27017
#bindIp: 127.0.0.1
bindIp: 0.0.0.0
```

2. Reiniciar servicio

Rainicia y comprueba que ha arrancado correctamente

```
sudo systemctl restart mongod
sudo systemctl status mongod
```

Configura el servidor para permitir tráfico entrante

Añade una regla en el servidor para permitir el tráfico entrante del puerto 27017. Para ello haz clic en la pestaña Seguridad y luego en el enlace de Grupos de seguridad

EC2 > Instancias

Instancias (1/1) Información

Buscar Instancia por atributo o etiqueta (case-sensitive)

Todos los estados

Estado de la instancia = running

Quitar los filtros

Instancias (1)

i-0193981641dd0ccbf En ejecución t3.micro 3/3 comprobacion Ver alarmas us-east-1c

Seguridad

Detalles Estado y alarmas Monitoreo Seguridad Redes Almacenamiento Etiquetas

Detalles de seguridad

Rol de IAM – ID del propietario 679480731783 Hora de lanzamiento Wed Nov 12 2025 10:30:41 GMT+0100 (hora estándar de Europa central)

Grupos de seguridad sg-02aa0093d9706111a (launch-wizard-1)

Entra en Reglas de entrada y haz clic en el botón Editar reglas de entrada

sg-02aa0093d9706111a - launch-wizard-1

Acciones

Detalles

Nombre del grupo de seguridad launch-wizard-1 ID del grupo de seguridad sg-02aa0093d9706111a Descripción launch-wizard-1 created 2025-10-12T11:32:38.739Z ID de la VPC vpc-062d7bf7b7a747a10

Propietario 679480731783 Número de reglas de entrada 2 Entradas de permisos Número de reglas de salida 1 Entrada de permiso

Reglas de entrada (2)

Reglas de salida Compartiendo : novedad Asociaciones de VPC : novedad Etiquetas

Name	ID de la regla del grupo	Versión de IP	Tipo	Protocolo	Intervalo de puertos	Origen	Descripción
-	sgr-0d0be3626cd926b6d	IPv4	SSH	TCP	22	0.0.0.0/0	-
-	sgr-0275101890de02829	IPv4	MYSQL/Aurora	TCP	3306	0.0.0.0/0	-

Administrar etiquetas Editar reglas de entrada

Haz clic en Agregar regla, configura el tipo, el puerto y la IP de origen 0.0.0.0/0 para permitir acceso desde cualquier lugar y por último haz clic en el botón Guardar reglas

Editar reglas de entrada Información

Las reglas de entrada controlan el tráfico entrante que puede llegar a la instancia.

ID de la regla del grupo de seguridad	Tipo	Información	Protocolo	Intervalo de puertos	Origen	Información	Descripción: opcional				
sgr-0d0be3626cd926b6d	SSH	TCP	22	Per...	0.0.0.0/0	X	Eliminar				
sgr-0275101890de02829	MYSQL/Aurora	TCP	3306	Per...	0.0.0.0/0	X	Eliminar				
-	1	2	TCP personalizado	3	27017	Any...	4	0.0.0.0/0	0.0.0.0/0	X	Eliminar
Agregar regla											

Advertencia: Las reglas cuyo origen es 0.0.0.0/0 o ::/0 permiten a todas las direcciones IP acceder a la instancia. Recomendamos configurar reglas de grupo de seguridad para permitir el acceso únicamente desde direcciones IP conocidas. X

Opciones: Cancelar, Previsualizar los cambios, Guardar reglas

En unos segundos aparecerá tu nueva regla en la lista

Las reglas del grupo de seguridad de entrada se han modificado correctamente en el grupo de seguridad (sg-02aa0093d9706111a | launch-wizard-1) X

► Detalles

sg-02aa0093d9706111a - launch-wizard-1 Acciones

Detalles		Reglas de entrada		Reglas de salida		Compartiendo : novedad		Asociaciones de VPC : novedad		Etiquetas	
Nombre del grupo de seguridad	launch-wizard-1	ID del grupo de seguridad	sg-02aa0093d9706111a	Descripción	launch-wizard-1 created 2025-10-12T11:32:38.739Z	ID de la VPC	vpc-062d7bf7b7a747a10				
Propietario	679480731783	Número de reglas de entrada	3 Entradas de permisos	Número de reglas de salida	1 Entrada de permiso						

Reglas de entrada (3)

Name	ID de la regla del grupo de seguridad	Versión de IP	Tipo	Protocolo	Intervalo de puertos	Origen	Descripción
-	sgr-0d0be3626cd926b6d	IPv4	SSH	TCP	22	0.0.0.0/0	-
-	sgr-0f3a26eea74e353f2	IPv4	TCP personalizado	TCP	27017	0.0.0.0/0	-
-	sgr-0275101890de02829	IPv4	MYSQL/Aurora	TCP	3306	0.0.0.0/0	-

Securizar MongoDB

1. Inicia el cliente (shell)

Ejecuta el comando siguiente:

```
mongosh
```

2. Crear usuario administrador

Conecta a la base de datos admin con el comando:

```
use admin
```

Crea el usuario con permisos sobre todas las bases de datos con el comando siguiente. Sustituye [usuario] y [contraseña] por los datos que quieras utilizar:

```
db.createUser({ user: "[usuario]", pwd: "[contraseña]", roles: [{ role: "root", db: "admin" }] })
```

Comprueba que se ha creado correctamente:

```
db.getUsers()

admin> db.getUsers()
{
  users: [
    {
      _id: '████████',
      userId: UUID('ae94f25d-7a16-4845-b76e-16e4aae67de2'),
      user: '████',
      db: '████',
      roles: [ { role: 'root', db: 'admin' } ],
      mechanisms: [ 'SCRAM-SHA-1', 'SCRAM-SHA-256' ]
    }
  ],
  ok: 1
}
```

3. Activa la autenticación

Sal del cliente (shell) y edita el fichero /etc/mongod.conf

```
sudo nano /etc/mongod.conf
```

Busca la sección security y descoméntala, luego añade la línea authorization: enabled.

```
security:
  authorization: enabled
```

Guarda los cambios de configuración, reinicia el servicio y comprueba que se ha iniciado correctamente:

```
sudo systemctl restart mongod
sudo systemctl status mongod
```

Comprueba que no puedes realizar operaciones entrando con mongosh :

```
:~$ mongosh
Current Mongosh Log ID: 6918da67f0376c53b29dc29c
Connecting to: mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.5.9
Using MongoDB: undefined
Using Mongosh: 2.5.9

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

test> show dbs
MongoServerError[Unauthorized]: Command listDatabases requires authentication
```

Sal del cliente (shell) y vuelve a entrar con el comando siguiente. Sustituye [usuario] por el usuario que has creado anteriormente y espera a que te pida la contraseña:

```
mongosh -u [usuario] -p --authenticationDatabase admin
```

Comprueba que ya puedes ver las bd con show dbs:

```

:~$ mongosh -u [REDACTED] -p --authenticationDatabase admin
Enter password: *****
Current Mongosh Log ID: 6918e215651fcb21d09dc29c
Connecting to:      mongodb://<credentials>@127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&authSource=admin&appName=mongosh+2.5.9
Using MongoDB:      8.2.1
Using Mongosh:      2.5.9

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

-----
   The server generated these startup warnings when booting
   2025-11-15T20:26:09.617+00:00: Using the XFS filesystem is strongly recommended with the WiredTiger storage engine.
   ee http://dochub.mongodb.org/core/prodnotes-filesystem
   2025-11-15T20:26:10.525+00:00: For customers running the current memory allocator, we suggest changing the contents
   f the following sysfsFile
   2025-11-15T20:26:10.525+00:00: For customers running the current memory allocator, we suggest changing the contents
   f the following sysfsFile
   2025-11-15T20:26:10.525+00:00: We suggest setting the contents of sysfsFile to 0.
-----

test> show dbs
admin   136.00 KiB
config  96.00 KiB
local   76.00 KiB

```

Crear base de datos y conecta desde Kotlin

1. Crear la BD

Ejecuta el comando siguiente:

```
use florabotanica
```

2. Añadir información

```
// Insertar documentos (si la colección no existe, se crea automáticamente)
db.plantas.insertMany([
    { id_planta: 1, nombre_comun: "Aloe", nombre_cientifico: "Aloe vera", altura: 30 },
    { id_planta: 2, nombre_comun: "Pino", nombre_cientifico: "Pinus sylvestris", altura: 330 },
    { id_planta: 3, nombre_comun: "Cactus", nombre_cientifico: "Cactaceae", altura: 120 }
])
```

3. Conectar desde Kotlin

Para conectar desde `Kotlin` necesitarás añadir la siguiente dependencia en al fichero `build.gradle.kts`

```
implementation("org.mongodb:mongodb-driver-sync:4.11.0")
```

Prueba el siguiente fragmento de código para comprobar que puedes acceder a la base de datos y mostrar la información que has insertado anteriormente. Sustituye `USUARIO` por tu nombre de usuario, `PASSWORD` por tu contraseña, `HOST` por el nombre o IP de tu servidor y `PUERTO` por 27017:

```

const val NOM_SRV = "mongodb://USUARIO:PASSWORD@HOST:PUERTO"
const val NOM_BD = "florabotanica"
const val NOM_COLECCION = "plantas"

import com.mongodb.client.MongoClients

fun mostrarPlantas() {
    val cliente = MongoClients.create(NOM_SRV)
    val db = cliente.getDatabase(NOM_BD)
    val colección = db.getCollection(NOM_COLECCION)

    // Mostrar documentos de la colección plantas
    val cursor = colección.find().iterator()
    cursor.use {
        while (it.hasNext()) {
            val doc = it.next()
            println(doc.toJson())
        }
    }
    cliente.close()
}

```

Autoría

Obra realizada por Begoña Paterna Lluch. Publicada bajo licencia [Creative Commons Atribución/Reconocimiento-CompartirIgual 4.0 Internacional](#)
