DAW
Desarrollo de Aplicaciones Web
2º Curso

DWES
Desarrollo Web Entorno Servidor

UD 5 Spring Boot

3. Spring DATA

IES BALMIS
Dpto Informática
Curso 2025-2026
Versión 6 (12/2025)

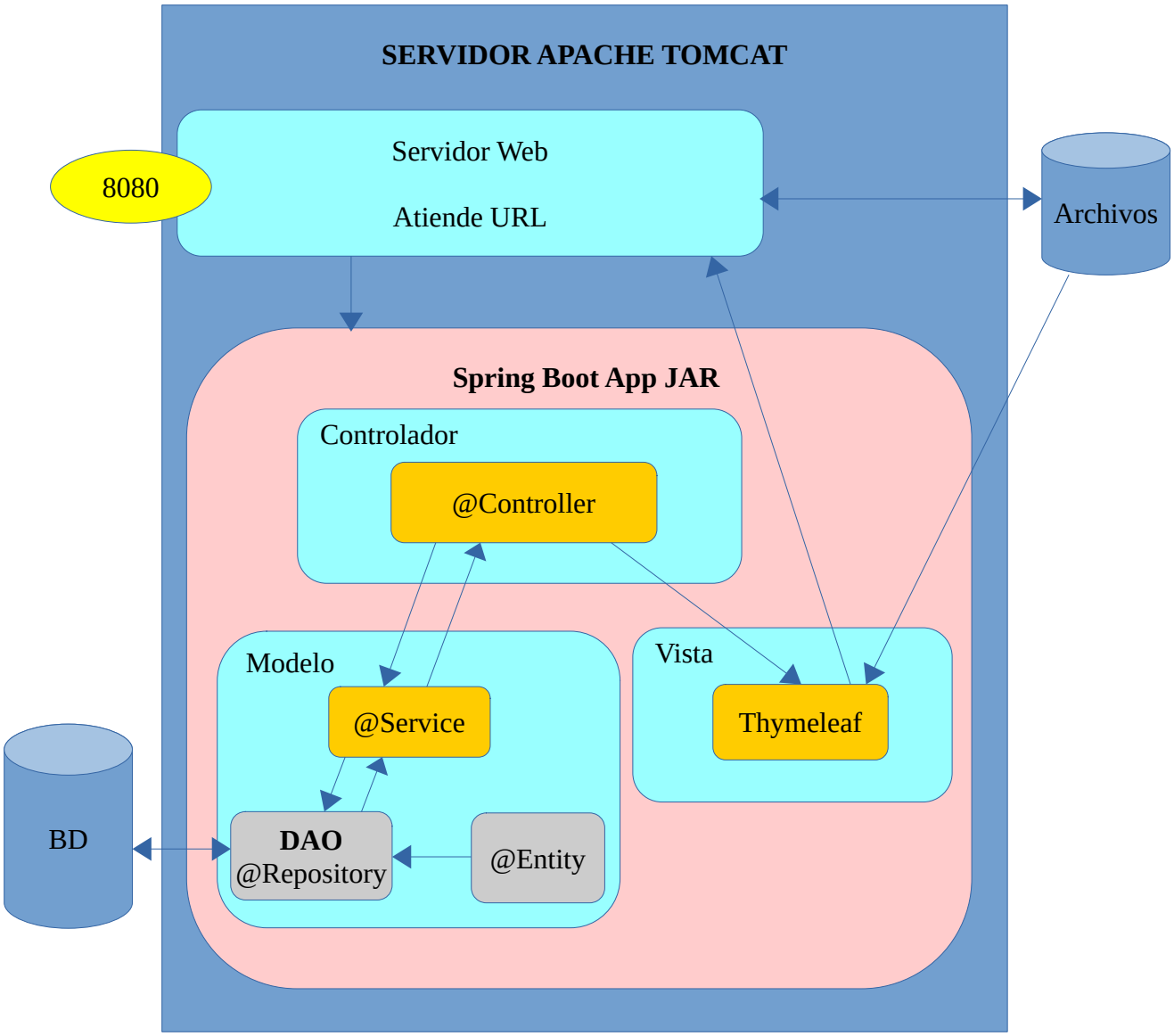
UD5.3 – Spring DATA

ÍNDICE

1. API con JSON

Siguiendo el patrón MVC podemos crear los diferentes componentes que nos permitan desarrollar proyectos Spring.

A los proyectos desarrollados hasta ahora, incorporaremos los componentes **@Repository** y **@Entity**



El nuevo componente con la anotación **@Repository** se encargará del acceso a datos, es decir, será el Objeto de Acceso a Datos (DAO – Data Access Object).

Este componente, si usa **JPA** necesitará que las clases de datos estén relacionadas con las tablas de las BD (Bases de Datos) mediante anotaciones **@Entity.**

Para simplificar y organizar nuestros primeros proyectos con DATA comenzaremos por:

- **no usar Thymeleaf**, sino crear proyectos API que muestren nuestros objetos en formato JSON utilizando **@ResponseBody** en **UserController**
- crear nuestras clases de datos de forma mucho más rápida utilizando **Lombok**
- usar **@Repository** para separar el acceso a datos de los servicios ofrecidos.

1.1 Proyecto user-list-v1

Para comprobar este nuevo funcionamiento, se ha copiado y modificado el proyecto **demo-thymeleaf-service-v3** para crear **user-list-v1.**

user-list-v1

```
-Cambiamos variables en el application.properties para el nuevo proyecto

-Eliminamos la dependencia de Thymeleaf y los archivos .html de templates

-Modificamos en UserController los métodos de tipo @ResponseBody y en vez de devolver
String devolvemos nuestros objetos o String en formato JSON, es decir, List<User> y User
respectivamente.

-Cargamos datos iniciales en la lista de usuarios de UserService con UserInitRunner
(estas clases, cuando usan objetos de Spring necesitan ser implements CommandLineRunner)
```

Descativar Thymeleaf

Para eliminar **Thymeleaf** de la configuración del proyecto de **Spring Boot**, eliminaremos la dependencia existente en el **pom.xml**.

pom.xml - Dependencias usadas en proyectos de Spring hasta ahora

```
<!-- SPRING WEB -->
<dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<!-- SPRING BOOT DEVTOOLS -->
<dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-devtools</artifactId>
     <scope>runtime</scope>
     <optional>true</optional>
</dependency>

<!-- THYMELEAF -->
<dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

Si en el proyecto se incorpora de forma automática **spring-boot-starter-webmvc** en vez de **spring-boot-starter-web**, se debe modificar ya que **spring-boot-starter-web** es más moderna e incluye el servidor embebido.

En estos proyectos donde creamos un API y devolvemos JSON podemos **eliminar la dependencia de Thymeleaf**.

Modificar el UserController

En la clase **UserController** se modifican los métodos para devolver contenido con **@ResponseBody**.

Spring usará la librerías de **Jackson** para mapear el contenido de los objetos **User** a **JSON** de forma automática.

1.2 Proyecto user-list-v2

Para continuar mejorando el proyecto, se ha copiado y modificado el proyecto **user-list-v1** para crear **user-list-v2.**

user-list-v2
-Cambiamos variables en el **application.properties** para el nuevo proyecto -Creamos la clase **@Repository** para separar el acceso a datos de las acciones a realizar -Cambiamos la clase de datos **User** incorporando las anotaciones de **Lombok** para ahorrarnos el escribir los métodos más habituales.

Uso de @Repository

Vamos a separar el modelo de MVC en 3 tipos clases:

- **@Service**: acciones a realizar que obtienen datos y los devuelven a la vista
- **@Repository**: métodos para el acceso a datos que podrá ser temporal en memoria RAM o persistente en bases de datos
- **Clases de datos**: con Lombok para simplificar la definición. Más adelante incorporaremos también @Entity para realizar la vinculación con las tablas de la base de datos

Creando la clase **@Repository** obtenemos:

Código más limpio	Crea **DataAccessException** automáticamente y no se necesita try-catch para: • SQLException, • jakarta.persistence.PersistenceException • org.hibernate.HibernateException • ...
Independencia de tecnología	Se puede cambiar de JPA a MongoDB sin cambiar el código de excepciones, es decir, la clase @Service NO cambia
Testing más fácil	Existen métodos que facilitan el testeo

Esta clase **@Repository** puede tener muchos métodos implementados, pero el **@Service** solo ofrece y atiende los que deseemos para cada aplicación.

Lombok

Las clases de datos con anotaciones de tipo Lombok permiten generar código estándar de forma automática o preimplementada

Project Lombok es una biblioteca para Java que nos ofrece nuevas funcionalidades. Para ello tiene que estar conectado a nuestro compilador, ya que su objetivo es facilitarnos el desarrollo de nuestro código evitándonos tener que escribir ciertos métodos, que van a ser repetitivos y que realmente tampoco aportan lógica al negocio.

El código que puede generar **Lombok** son métodos como **getters/setters, equals, constructores** que nos facilitan el mantenimiento y la limpieza del código.

Las anotaciones más usadas son:

```
@AllArgsConstructor         // => Constructor con todos los argumentos
@NoArgsConstructor          // => Constructor sin argumentos
@RequiredArgsConstructor
        // => Constructor cuando tenemos campos nulos o final
        // => No se incluye si existe @AllArgsConstructor y/o @NoArgsConstructor
@Getter
@Setter
@ToString
@EqualsAndHashCode
@Data   // => @Getter + @Setter + @ToString + @EqualsAndHashCode + @RequiredArgsConstructor
```

Para que funcionen estas anotaciones añadiremos la siguiente dependencia.

pom.xml - lombok

```
<!-- SPRING LOMBOK -->
<dependency>
     <groupId>org.projectlombok</groupId>
     <artifactId>lombok</artifactId>
     <optional>true</optional>
</dependency>
```

En NetBeans funciona a partir de la versión 4.0.0.
En las versiones 3.x.x no reconoce las anotaciones de Lombok.

1.3 Proyecto user-list-v3

Para continuar mejorando el proyecto, se ha copiado y modificado el proyecto **user-list-v2** para crear **user-list-v3**.

user-list-v3

-Cambiamos variables en el **application.properties** para el nuevo proyecto

-Devolver objeto en **Controller** de tipo **Map<String, Object>** para que sea mapeado a JSON automáticamente

-Mejorar diseño **static** con **index.html y pruebas.html**

Controller de tipo Map<String, Object>

En el método **count**, en vez de montar el String de JSON manualmente, creamos una estructura **Map<String, Object>** para que sea Spring el que convierta su contenido a JSON.

UserController

```
    @GetMapping("/users/count")
    public Map<String, Object> countUsers() {

        Map<String, Object> obj = new HashMap<>();
        obj.put("users", userService.count());

        return obj;  // Se mapea automáticamente a JSON usando Jackson
    }
```

index.html y pruebas.html

En la carpeta **static** se han incorporado dos archivos **html** con diseño utilizando **bootstrap** para probar los métodos del API.

1.4 Resumen

Resumen de características incluidas en un proyecto **Spring** de tipo **API-MVC con Datos**:

- No incluir **Thymeleaf** porque devolvemos datos en JSON
- Incluir clase **@Repository** para separar en el MODELO métodos del API de métodos de acceso a datos
- Incluir clases de datos con **Lombok** para simplificar
- Incluir clase de gestión de errores **Global** en Java Package **exception**

Para facilitar la comprensión de los conocimientos de los diferentes componentes que vamos a ir usando, usaremos unos iconos:

✅	-Uso recomendable para API en Spring -Uso recomendable en los ejercicios propuestos
❌	-Uso no recomendable para API en Spring -Existen otros componentes, clases, métodos o formas de realizar los mismo de forma más adecuada y útil para nuestro propósito **-No usar en el examen**
	Incluido en proyectos anteriores
	Cambio en este proyecto pero existen otras formas más adecuadas
	Incluido en proyectos anteriores Cambio o propuesta de uso recomendable (SE DEBE USAR EN LOS EJERCICIOS Y EXAMEN)

A continuación mostramos las características incluidas en cada proyecto:

	PROYECTOS		
	list-v1	**list-v2**	**list-v3**
Thymeleaf	❌	❌	❌
Lombok en clases de datos	❌	✅	✅
Controlador	❌ **@ResponseBody**	❌ **@ResponseBody**	✅ **@RestController** **Map<String, Object>**
Service con Repository	❌	✅	✅
Documentación en HTML	❌	❌	✅
Acceso a datos	❌ **Service**	✅ **Repository**	✅ **Repository**
Base de Datos	❌	❌	❌
Persistencia	❌ **ArrayList RAM**	❌ **ArrayList RAM**	❌ **ArrayList RAM**
Carga de Datos inicial	❌ **CommandLineRunner**	❌ **@PostConstruct**	❌ **@PostConstruct**

2. Acceso a Bases de Datos con JDBC

Hasta ahora hemos usado un **ArrayList** de objetos para almacenar la información en memoria **RAM** usando la clase **@Repository**, pero en los siguientes proyectos usaremos bases de datos de tipo relacional, es decir, **SQL**.

2.1 Proyecto user-h2-jdbc-v1

Para continuar mejorando el proyecto, se ha copiado y modificado el proyecto **user-list-v3** para crear **user-h2-jdbc-v1**.

user-h2-jdbc-v1

-Añadir nuevas dependencias
- **spring-boot-starter-jdbc**
- **h2**
- **spring-boot-h2console**

-Cambiamos variables en el **application.properties** para el nuevo proyecto

-Adaptamos el **application.properties** para usar **H2 Database** en **RAM** sin persistencia

-Usar **Datasource** en la clase **@Repository** con SQL para el acceso a datos mediante objetos de tipo Connection, Statement y ResultSet, que permiten obtener los datos almacenados en H2 Database.

-Adaptar en la clase **@Repository** la carga inicial en **@PostConstruct** utilizando **Datasource** para realizar el CREATE TABLE y los INSERT controlando si ya existen datos.

-Adaptar en la clase **@Repository** todos los métodos **Datasource**

-La carga inicial la realizamos utilizando dos archivos, schema.sql y data.sql

-Gestión de errores (**package exception**) a nivel global

Nuevas dependencias

Spring-boot-starter-jdbc proporciona las clases **Datasource y JdbcTemplate** para poder conectar usando **JDBC** y ejecutar instrucciones SQL sobre las bases de datos.

H2 instala el driver **JDBC** para usar bases de datos **H2 Database**.

Spring-boot-h2console añade una URL accesible para manipular (como phpMyAdmin en MySQL) los datos de una base de datos **H2 Database**.

pom.xml – H2 Database

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

<!-- H2 DATABASE -->
<dependency>
     <groupId>com.h2database</groupId>
     <artifactId>h2</artifactId>
     <scope>runtime</scope>
</dependency>
<dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-h2console</artifactId>
</dependency>
```

application.properties

En nuestro primer ejemplo usaremos una Base de Datos H2 en memoria RAM.
Para usar una Base de Datos es necesario configurar una serie de variables que usará Spring para realizar la conexión.

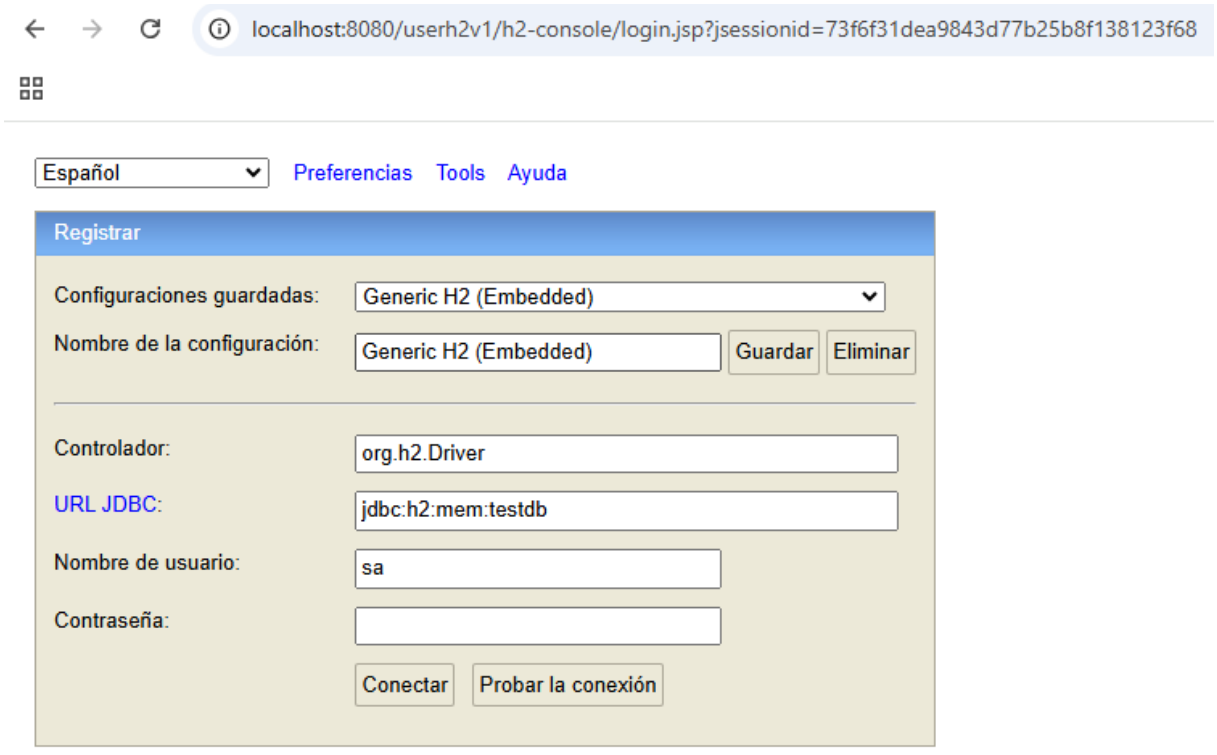
application.properties para H2 en RAM

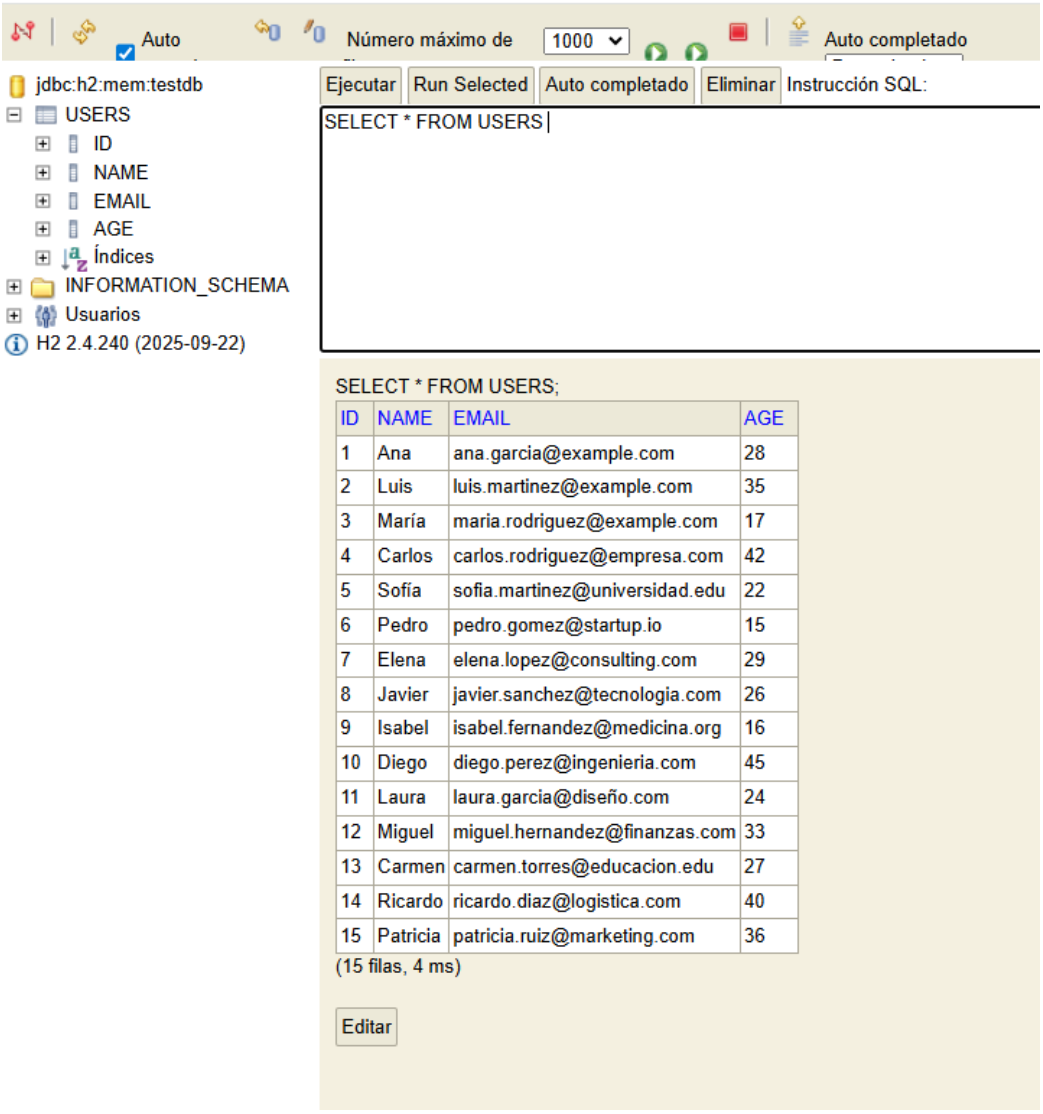
```
# Configuración de la base de datos (H2 en memoria)
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

# H2 Console (accesible en http://localhost:8080/.../h2-console )
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
spring.h2.console.settings.web-allow-others=true
spring.h2.console.settings.trace=false
```

¡¡CUIDADO!!
En el archivo **application.properties** es MUY IMPORTANTE no añadir espacios al final de las líneas.

La url de conexión **jdbc:h2:mem:testdb** le indica a H2 Database que se usará en memoria (**mem**) una BD denominada **testdb**.

La dependencia **h2console** permite mostrar una consola para manipular los datos al estilo de **phpMyAdmin** para MySQL.





DataSource

El **DataSource** funciona con la conexión **JDBC** indicada en **application.properties** sin necesidad de ninguna dependencia adicional de data.

Con **spring-boot-starter-jdbc** se consiguen:

- Utilidades JDBC de Spring
- No necesita repositorios pero es recomendable
- No necesita mapeo automático
- Se escribe todo el SQL manualmente

En la clase **UserRepository** se han escrito los métodos que permiten ejecutar instrucciones SQL con **Datasource** para acceder a la BD.

Este ejemplo de uso de **Datasource** solo sirve para explicar cómo funciona internamente Spring Boot, ya que usaremos otros recursos más cómodos y adecuados para el uso del acceso a datos.

Gestión de errores con GlobalExceptionHandler

La anotación **@RestControllerAdvice** nos permite gestionar de forma global en el proyecto cualquier error que se produzca.

Con la anotación **@ExceptionHandler(XXXXXXXXXXXXX.class)** podemos capturar cualquier error como se haría en un **try-catch** pero con la ventaja de que lo tenemos centralizado en una clase.

Como tenemos un API que muestra datos en JSON lo que hacemos es crear un **Map<String, Object>** con los datos más importantes del error que será mapeado automáticamente por las librerías de Jackson a JSON.

Los errores capturados con número de **Response.status** devuelto son:

- **ResourceNotFoundException**: Recurso no encontrado (404 personalizado)
- **NoHandlerFoundException**: URL no encontrada (404)
- **NoResourceFoundException**: Error de recurso estático (convertir a 404)
- **MethodArgumentTypeMismatchException**: Error de tipo de parámetro (400)
- **MissingServletRequestParameterException**: Faltan parámetros obligatorios (400)
- **Exception**: resto de errores (500)

Gestión de errores con @Repository

Hemos incluido la gestión de errores incluida por **@Repository** con las dependencias:

- **`spring-boot-starter-data-jdbc`** `(utilizada en estos proyectos)`
- **`spring-boot-starter-data-jpa`** `(utilizada más adelante)`

Con obtenemos una jerarquía clara de errores:

DataAccessException
- **DataIntegrityViolationException (UNIQUE, FOREIGN KEY)**
- **EmptyResultDataAccessException (no encontrado)**
- **CannotGetJdbcConnectionException (BD caída)**
- **BadSqlGrammarException (SQL con errores)**
- **OptimisticLockingFailureException (concurrencia)**

GlobalExceptionHandler

```
    @ExceptionHandler(DataIntegrityViolationException.class)
    public ResponseEntity<?> handleDuplicate() {
        return ResponseEntity.status(409).body("Duplicate entry");
    }

    @ExceptionHandler(EmptyResultDataAccessException.class)
    public ResponseEntity<?> handleNotFound() {
        return ResponseEntity.status(404).body("Not found");
    }

    // Antes de capturar EL RESTO de errores con Exception.class
```

2.2 Proyecto user-h2-jdbc-v2

Para continuar mejorando el proyecto, se ha copiado y modificado el proyecto **user-h2-jdbc-v1** para crear **user-h2-jdbc-v2**.

user-h2-jdbc-v1

-Cambiamos variables en el **application.properties** para el nuevo proyecto

-Adaptamos el **application.properties** para usar **H2 Database** con persistencia en un **fichero** del disco duro

-Usar **JdbcTemplate** en la clase **@Repository** con SQL para el acceso a datos que permiten obtener los datos almacenados en H2 Database, lo que simplifica la sintaxis para un mejor mantenimiento, ya que nos ofrece directamente el ResultSet con los datos.

-Adaptar en la clase **@Repository** la carga inicial en **@PostConstruct** utilizando **JdbcTemplate** para realizar el CREATE TABLE y los INSERT controlando si ya existen datos.

-Adaptar en la clase **@Repository** todos los métodos **JdbcTemplate**

application.properties

Para usar un fichero con los datos solo es necesario cambiar la URL de conexión.

application.properties para H2 en fichero

```
# Configuración de la base de datos (H2 en memoria)
spring.datasource.url=jdbc:h2:file:./datos/users;DB_CLOSE_ON_EXIT=FALSE;AUTO_RECONNECT=TRUE
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

# H2 Console (accesible en http://localhost:8080/.../h2-console )
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
spring.h2.console.settings.web-allow-others=true
spring.h2.console.settings.trace=false
```

La url de conexión **jdbc:h2:file:./datos/users** le indica a H2 Database que se usará en memoria (**mem**) una BD denominada **users** en la carpeta **./datos** que será creada si no existe.

JdbcTemplate

El **JdbcTemplate,** al igual que el **DataSource,** funciona con la conexión **JDBC** indicada en **application.properties** sin necesidad de ninguna dependencia adicional de data.

Con **spring-boot-starter-jdbc** se consiguen:

- Utilidades JDBC de Spring
- **JdbcTemplate** y NamedParameterJdbcTemplate
- **JdbcTemplate** es más cómodo que **DataSource**
- Soporte para transacciones (@Transactional)
- No necesita repositorios pero es recomendable
- No necesita mapeo automático
- Se escribe todo el SQL manualmente

En la clase **UserRepository** se han escrito los métodos que permiten ejecutar instrucciones SQL con **JdbcTemplate** para acceder a la BD.

Este ejemplo de uso de **JdbcTemplate** solo sirve para explicar cómo funciona internamente Spring Boot, ya que usaremos otros recursos más cómodos y adecuados para el uso del acceso a datos.

2.3 Proyecto user-mysql-jdbc-v3

Para continuar mejorando el proyecto, se ha copiado y modificado el proyecto **user-h2-jdbc-v2** para crear **user-mysql-jdbc-v3**.

user-mysql-jdbc-v3

-Cambiamos variables en el **application.properties** para el nuevo proyecto

-Cambiamos las **dependencias en el pom.xml** para usar **MySQL** en vez de H2 Database

-Adaptamos el **application.properties** para usar MySQL

-Carga inicial de datos:
- Creamos dos archivos **schema.sql y data.sql** en **/src/main/resources** con las instrucciones de CREATE TABLE y los INSERT respectivamente para la carga inicial de datos.
- Adaptamos el **application.properties** con la variable **spring.sql.init.mode** para controlar el comportamiento de la ejecución de los script .sql
- Eliminamos de la clase **@Repository** el método **@PostConstruct**

Dependencias necesarias

Spring-boot-starter-jdbc proporciona las clases **Datasource y JdbcTemplate** para poder conectar usando **JDBC** y ejecutar instrucciones SQL sobre las bases de datos.

Mysql-connector-j instala el driver **JDBC** para usar bases de datos **H2 Database**.

Eliminar o comentar las dependencias de H2

pom.xml – MySQL

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

application.properties para usar MySQL

Para usar un fichero con los datos solo es necesario cambiar la URL de conexión.

application.properties para MySQL

```
# Configuración de conexión MySQL
spring.datasource.url=jdbc:mysql://localhost:3306/dawbd?
createDatabaseIfNotExist=true&useSSL=false&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

La url de conexión **jdbc:mysql:...** puede contener más parámetros para configurar la conexión.

Si la instalación de MySQL requiere contraseña para el usuario se indicará en la variable **spring.datasource.password**.

Para probar este proyecto será necesario tener en marcha una instalación de **XAMPP** portable como las utilizadas con PHP.

application.properties para la carga inicial

Se añaden los archivos .sql en **/src/main/resources** y se actualiza **application.properties** para que se ejecuten si no existe la BD con las tablas.

application.properties

```
# Carga inical de datos usando archivos .sql
# => always     =>siempre
# => embedded   =>solo para H2 en memoria
# => never      =>nunca
spring.sql.init.mode=always
spring.sql.init.continue-on-error=true
```

La variable **spring.sql.init.mode=always** le indica a **Spring Boot** que ejecute los archivos **schema.sql** y **data.sql** siempre.

- Fuerza a Spring Boot a ejecutar siempre los scripts de inicialización SQL # (schema.sql y data.sql), incluso en bases de datos no embebidas.
- Por defecto, solo se ejecutan en BD embebidas (como H2).

La variable **spring.sql.init.continue-on-error=true** le indica a **Spring Boot** que continue aunque alguna instrucción de los scripts .sql de error. Esto permitiría continuar aunque los INSERT produzcan error por vilación de la clave principal.

De todas formas, como el valor **embedded** no funciona con MySQL y aunque hayamos puesto **spring.sql.init.continue-on-error=false**, en MyQL podemos modificar las instrucciones **INSERT INTO** por **INSERT IGNORE INTO** de forma que al repetirse el id ignore el error y continue.

Si usamos H2 en memoria, podemos seguir con las instrucciones INSERT usando el ID automático y sin IGNORE.

2.4 Resumen

A continuación mostramos las características incluidas en cada proyecto:

	PROYECTOS		
	h2-jdbc-v1	**h2-jdbc-v2**	**mysql-jdbc-v3**
Thymeleaf	❌	❌	❌
Lombok	✅	✅	✅
Controlador	✅ **@RestController Map<String, Object>**	✅ **@RestController Map<String, Object>**	✅ **@RestController Map<String, Object>**
Service con Repository	✅	✅	✅
Documentación HTML	✅	✅	✅
GlobalExceptionHandler	✅	✅	✅
Acceso a Datos	❌ **JDBC Datasource SQL**	❌ **JDBC JdbcTemplate SQL**	❌ **JDBC JdbcTemplate SQL**
Base de Datos	✅ **H2 Database**	✅ **H2 Database**	✅ **MySQL**
Persistencia	✅ RAM	✅ FILE	✅ SERVER
Carga de Datos inicial	❌ **@PostConstruct Datasource**	❌ **@PostConstruct JdbcTemplate**	✅ **shema.sql + data.sql**

3. Acceso a Bases de Datos con JPA

Hasta ahora hemos usado un **JDBC** para el acceso a datos usando la clase **@Repository**, pero en los siguientes proyectos usaremos **JPA** que nos ofrecerá mayor funcionalidad y simplicidad con bases de datos de tipo relacional, es decir, **SQL**.

3.1 Proyecto user-h2-jpa-v1

Para continuar mejorando el proyecto, se ha copiado y modificado el proyecto **user-mysql-jdbc-v3** para crear **user-h2-jpa-v1**.

user-h2-jpa-v1

-Eliminar dependencias
- **spring-boot-starter-jdbc**

-Añadir nuevas dependencias
- **spring-boot-starter-data-jpa**
- **spring-boot-starter-validation**

-Cambiamos variables en el **application.properties** para el nuevo proyecto

-Añadir nuevas variables en el **application.properties** para configurar JPA.

-Usar **JpaRepository** en la clase **@Repository** con SQL para el acceso a datos con la anotación **@Query** y el parámetro **nativeQuery = true** lo que simplifica y facilita el desarrollo y manteniemiento.

-Añadir en las **clases de datos** (User) de **model**, las anotaciones de **jakarta.persistence** y **jakarta.validation**, para que JPA realice tanto el mapeo automático como las validaciones.

Nuevas dependencias

Spring-boot-starter-data-jpa proporciona la clase **JpaRepository** para poder conectar usando **JDBC** y ejecutar instrucciones SQL yJPQL sobre las bases de datos realizando mapeos automáticos.

Spring-boot-starter-validation implementa una serie de **anotaciones** para validar los datos introducidos en un objeto de la clase de datos (User).

Eliminar o comentar la dependencia de Spring-boot-starter-jdbc

pom.xml – DATA-JPA

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

spring-boot-starter-data-jpa agrega:

- Incluye todo lo de spring-boot-starter-jdbc (transitivo)
- Hibernate (implementación JPA)
- Spring Data JPA repositories
  - **JpaRepository**
- Tendremos clases con @Entity
- Usaremos @Repository con JpaRepository
- Usaremos mapeo objeto-relacional (ORM)
- Utilizaremos SQL/JPQL
- Dispondremos de transacciones automáticas

application.properties para usar JPA

Para usar un fichero con los datos solo es necesario cambiar la URL de conexión.

application.properties para JPA

```
# Configuración de conexión JPA
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=validate
spring.jpa.show-sql=true
spring.jpa.defer-datasource-initialization=false
```

spring.jpa.database-platform: Especifica el dialecto de base de datos que Hibernate debe usar.

- **org.hibernate.dialect.H2Dialect**: para H2
- **org.hibernate.dialect.MySQL8Dialect**: para MySQL

spring.jpa.show-sql:
-Controla si se muestran las sentencias SQL generadas por Hibernate.
-Muestra en la consola las sentencias SQL generadas por Hibernate (JPA).
-Ayuda a depurar consultas, pero no muestra los valores de los parámetros.

- **true**: recomendable en desarrollo
- **false**: recomendable en producción

spring.jpa.hibernate.ddl-auto:
-Controla cómo Hibernate maneja el esquema de la base de datos.
-Crea el esquema de la base de datos al arrancar la aplicación y lo elimina al detenerla.
-Útil solo en desarrollo o pruebas (¡NUNCA en producción con create ni update!).

- **none**: No hacer nada
- **update**: Actualizar el esquema si es necesario
- **create**: Crear al inicio, pero no eliminar al cerrar
- **create-drop**: Crear al inicio, pero no eliminar al cerrar
- **validate**: Validar que el esquema coincide con las entidades

spring.jpa.defer-datasource-initialization:
-Controla cuándo se inicializa la base de datos, según lo indicado en la variable spring.jpa.hibernate.ddl-auto.
-Consiste en diferir la inicialización de datos (como data.sql) hasta después de que Hibernate haya creado las tablas (útil cuando usas JPA + data.sql juntos).
-Evita errores como "tabla no existe" al cargar datos iniciales.

- **true**: La inicialización del esquema (creación de tablas) se realiza antes de que se carguen los datos iniciales (como los del archivo data.sql).
- **false**: Se cargan los datos después de que la aplicación esté completamente lista.
- **Importante**: Con create o create-drop, si esto fuera false, los datos de data.sql se perderían inmediatamente porque las tablas se eliminan al final.

Uso de JpaRepository

En esta versión usaremos las consultas SQL mapeadas mediante anotación **@Query** y el parámetro **nativeQuery = true,** indicando la instrucción **SQL** a mapear.

En el método decidiremos el tipo de objeto que deseamos recoger para que JPA lo mapee automáticamente.

Anotaciones en clases de datos del Java Package model

Además las anotaciones usadas de **Lombok**, ahora añadiremos las de **jakarta.persistence** para relacionar nuestras clases con las tablas de la base de datos.

Las principales anotaciones de clase son:

@Entity
Indica que es una clase de datos mapeada con una tabla de una BD
@Table(name = "xxxxx")
Indica la tabla de la BD con la que está relacionada

Además, tendrán para cada campo o propiedad de una clase disponemos de otras anotaciones como:

@Column(name = "XXXX", nullable = false, unique = false)
Cuál es la columna correspondiente en la tabla que le representa. También se puede indicar (es opcional) si el valor del campo puede ser nulo o si el valor del campo no se puede repetir.
@Id
Indica el campo clave
@GeneratedValue(strategy = GenerationType.IDENTITY)
Campo que en la BD es AUTO_INCREMENT

Ya para validar o comprobar los valores de las propiedades también podemos añadir otras anotaciones de **jakarta.validation** como

`@NotBlank(message = "El nombre no puede estar vacío")`
Para comprobar un campo que no puede estar vacío
`@Size(min=1, max = 255, message = "El título no puede tener más de 255 caracteres")`
Para comprobar la cantidad de caracteres mínima y/o máxima que admitimos
`@Min(value = 0, message = "La edad mínima es 0")` `@Max(value = 150, message = "La edad máxima es 150")`
Para comprobar el valor mínimo y/o máximo que admitimos

Dependencia DATA-JDBC (NO usar, mejor DATA-JPA)

Spring también ofrece la dependencia **DATA-JDBC** con el siguiente código Maven para el pom.xml:

pom.xml – DATA-JDBC

```
<!-- SPRING DATA-JDBC -->
<dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-data-jdbc</artifactId>
</dependency>
```

spring-boot-starter-data-jdbc agrega:

- Todo lo de spring-boot-starter-jdbc (transitivo)
- Un enfoque diferente de mapeo ORM
- Spring Data JDBC repositories
  - **CrudRepository**
- No necesitamos usar @Entity (usas @Table y @Id de Spring Data JDBC)
- No queremos ORM complejo
- Queremos un mapeo más simple y directo
- Preferimos usar SQL nativo

Si en vez de usar **CrudRepository** usamos **JpaRepository** tendremos:

- Más útil: Retorna List en lugar de Iterable
- Más funcional: Paginación y ordenación nativas
- Más eficiente: Operaciones en BD, no en memoria
- Más estándar: Ideal para APIs REST paginadas

Nosotros NO USAREMOS **CrudRepository.**
Usaremos **JpaRepository** porque es mucho más potente.

3.2 Proyecto user-h2-jpa-v2

Para continuar mejorando el proyecto, se ha copiado y modificado el proyecto **user-h2-jpa-v1** para crear **user-h2-jpa-v2**.

Además de poder mapear de forma automática cualquier instrucción SQL sobre la base de datos como hemos visto en el proyecto anterior, JpaRepository ofrece una serie de métodos disponibles que vienen heredados. Por ejemplo:

De consulta:

findAll()	Obtener todos los registros
findById(id)	Obtener el registro filtrando por el id
count()	Contar el número de registros

De comprobación:

equals(User)	Comprobar si un registro es igual a otro
exist(User)	Comprobar si existe un registro
existById(id)	Comprobar si existe un registro filtrando por el id

De actualización:

delete(User)	Eliminar un registro
deleteById(id)	Eliminar un registro filtrando por el id
deleteAll()	Eliminar todos los registros
save(User)	Grabar un registro

Además de estos métodos heredados, JpaRepository nos ofrece la posibilidad de crear nuevos métodos de forma automática a través del nombre. A esta tecnología se la denomina **Métodos de Consulta Derivados (DQM = Derived Query Methods)**.

Para crear métodos DQM necesitamos una serie de componentes:

Verbo*	Acción obligatoria: **`find`**, `read`, `get`, `query`, `search`, `stream`	`find...`
Por*	Palabra obligatoria después del verbo: **`By`**	`findBy...`
Propiedad*	Nombre obligatorio del campo en la entidad (con mayúscula inicial) **`Email, Name, CreatedAt,`** etc.	`findByEmail`
Operador	Condición a cumplir (opcional): **`Is`, `Equals`, `IsNull`, `Like`, `GreaterThan`,** etc.	`findByAgeGreaterThan`
Conjunción	Para unir varias condiciones (opcional): `And`, `Or`	`findByEmailAndName`
Orden	`OrderBy[Propiedad][Asc/Desc]`	`FindByNameOrderByAgeDesc`
find es el verbo más común, pero read, get, query funcionan igual.		

La lista de operadores soportados es:

Is, Equals	`findByEmail(String email)`	`WHERE email = ?`
IsNull	`findByEmailIsNull()`	`WHERE email IS NULL`
IsNotNull, NotNull	`findByEmailIsNotNull()`	`WHERE email IS NOT NULL`
Like	`findByNameLike(String name)`	`WHERE name LIKE ?`
NotLike	`findByNameNotLike(String name)`	`WHERE name NOT LIKE ?`
StartingWith	`findByNameStartingWith(String prefix)`	`WHERE name LIKE ?%`
EndingWith	`findByNameEndingWith(String suffix)`	`WHERE name LIKE %?`
Containing	`findByNameContaining(String fragment)`	`WHERE name LIKE %?%`
GreaterThan	`findByAgeGreaterThan(int age)`	`WHERE age > ?`
LessThan	`findByAgeLessThan(int age)`	`WHERE age < ?`
Between	`findByAgeBetween(int min, int max)`	`WHERE age BETWEEN ? AND ?`
In	`findByIdIn(Collection<Long> ids)`	`WHERE id IN ?`
NotIn	`findByIdNotIn(Collection<Long> ids)`	`WHERE id NOT IN ?`
True / False	`findByActiveTrue()`	`WHERE active = true`
IgnoreCase	`findByNameIgnoreCase(String name)`	`WHERE UPPER(name) = UPPER(?)`

También se pueden combinar:

```
findByEmailContainingAndAgeGreaterThan(String email, int age)
```

En este proyecto de ejemplo usaremos DQM en vez de SQL para el acceso a datos de la clase @Repository.

user-h2-jpa-v2

-Cambiamos variables en el **application.properties** para el nuevo proyecto

-Usar **JpaRepository** en la clase **@Repository** con **DQM** para el acceso a datos.

Cambios en el UserRepository

Hemos sustituido los métodos que usan **SQL** con métodos creados mediante el nombre utilizando **DQM (Derived Query Methods)**.

3.3 Proyecto user-h2-jpa-v3

Para continuar mejorando el proyecto, se ha copiado y modificado el proyecto **user-h2-jpa-v2** para crear **user-h2-jpa-v3**.

En este proyecto usaremos JQPL en UserRepository para el acceso a datos, sin necesidad de usar SQL ni DQM.

JPQL es un lenguaje parecido a SQL pero que consulta los datos teniendo en cuenta el mapeo, es decir, las clases de datos sin tener en cuenta cómo estás definidas en las tablas de la BD.

Para poder hacer consultas, en la misma instrucción se realiza una instancia de la clase. Por ejemplo:

```
SELECT usu FROM User usu
```

Como vemos, en SQL la tabla se llama users, mientras que nuestra clse es User. Además instanciamos un objeto usu de la clase para filtrar sobre sus propiedades. Por ejemplo:

```
SELECT usu FROM User usu WHERE usu.id > 7
```

Para profundizar en **JPQL** se puede consultar el siguiente enlace:

JPA – JPQL en tutorialspoint.com
https://www.tutorialspoint.com/jpa/jpa_jpql.htm

user-h2-jpa-v3
-Cambiamos variables en el **application.properties** para el nuevo proyecto -Usar **JpaRepository** en la clase **@Repository** con **JPQL** para el acceso a datos.

Cambios en el UserRepository

Hemos sustituido los métodos que usan **SQL** con métodos creados mediante lenguaje **JPQL**.

3.4 Resumen

Versiones con JpaRepository

- Usando SQL para simplificar y comprender mejor qué está haciendo
- Usando JPQL
- Usando DQM

A continuación mostramos las características incluidas en cada proyecto:

	PROYECTOS		
	h2-jpa-v1	**h2-jpa-v2**	**h2-jpa-v3**
Thymeleaf	❌	❌	❌
Lombok	✅	✅	✅
Controlador	✅ **@RestController Map<String, Object>**	✅ **@RestController Map<String, Object>**	✅ **@RestController Map<String, Object>**
Service con Repository	✅	✅	✅
GlobalExceptionHandler	✅	✅	✅
Documentación HTML	✅	✅	✅
Acceso a Datos	✅ **JpaRepository SQL**	❌ **JpaRepository DQM**	❌ **JpaRepository JPQL**
Base de Datos	✅ **H2 Database**	✅ **H2 Database**	✅ **H2 Database**
Persistencia	✅ FILE	✅ FILE	✅ FILE
Carga de Datos inicial	✅ **shema.sql + data.sql**	✅ **shema.sql + data.sql**	✅ **shema.sql + data.sql**

4. JPA con varias tablas

Cuando tenemos varias tablas relacionadas, podemos usar las relaciones existentes para mostrar datos utilizando los campos FOREIGN KEY que tengamos.

En estos proyectos tendremos en cuenta varios aspectos:

- Un controlador, servicio, repositorio, modelo para cada tabla
- Mostrar todos los datos o simplemente algún dato de la tabla relacionada
- Evitar la recursividad, es decir, solo mostrar los datos de la tabla a primer nivel pero no seguir buscando datos relacionados con la siguiente tabla.

En los proyectos de ejemplo se han creado dos tablas, **USERS** y **ROLES**, de forma que un **usuario** tiene un **rol** (relación muchos a uno) y un rol tiene muchos usuarios (relación uno a muchos).

En los siguientes proyectos usaremos todo lo recomendado, es decir, copiaremos el proyecto **user-h2-jpa-v1** a otro **user-h2-jpa2-v1**. Recordamos por lo tanto usaremos:

- **No usar Thymeleaf** para crear el API
- Usar anotaciones **Lombok**
- Controlador con **@RestController y objetos Map<String, Object>**
- Service con **Repository**
- **GlobalExceptionHandler** para la gestión de errores
- Documentación HTML
- Acceso a Datos usando **JpaRepository** con **SQL**
- Uso de **H2** como Base de Datos
- Uso de **fichero** para la persistencia
- Carga de datos inicial con **schema.sql y data.sql**

4.1 Proyecto user-h2-jpa2-v1

user-h2-jpa2-v1

-Cambiamos variables en el **application.properties** para el nuevo proyecto

-Actualizamos **schema.sql y data.sql** con las tablas y datos iniciales que incluyen **USUARIOS** y **ROLES**

-Crear la clase **Role** y los componentes **RoleController**, **RoleService** y **RoleRepository**

-Añadir a las clases de datos los **campos de las relaciones**.

-Añadir anotaciones para **evitar la recursividad**

Roles

Hemos creado la clase **Role** utilizando las mismas anotaciones que en User.
Hemos duplicado y adaptado los nuevos componentes **RoleController**, **RoleService** y **RoleRepository**.

Campos de las relaciones

La anotación más importante es la que indica el tipo de relación existente entre las dos tablas, y pueden ser:

@OneToOne @OneToMany @ManyToOne @ManyToMany

User - Añadimos

```
...
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "role_id", referencedColumnName = "id")
    private Role role;
...
```

Role - Añadimos

```
...
    @OneToMany(mappedBy = "role", cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    private Set<User> users = new HashSet<>();

    // Método para obtener el número de usuarios
    @JsonProperty("userCount")
    public int getUserCount() {
        return users != null ? users.size() : 0;
    }
...
```

@ManyToOne de User

El valor **FetchType.LAZY** (Carga Perezosa) es el recomendado ya que usa una estrategia de carga que significa "no cargar hasta que sea necesario".

@JoinColumn

Esta anotación usa los datos indicados en el **FOREIGN KEY de la instrucción CREATE TABLE**, es decir, los dos campos que deben coincidir del **JOIN**. Primero el campo de la clase en la que estamos (users.**role_id**) y el segundo el campo de la tabla relacionada (roles.**id**).

@OneToMany de Role

En el caso de la clase Role añadimos el atributo **mappedBy = "role"** para indicar que la relación está definida ya con **@JoinColumn** en la propiedad "**role**" de la clase User, que es a la que pertenece la propiedad

private Set<**User**> users = new HashSet<>();

Indicamos también con **cascade = CascadeType.ALL** que las operaciones que hacemos en una entidad se propagarán automáticamente a las entidades relacionadas.

El valor **FetchType.LAZY** (Carga Perezosa) es el recomendado ya que usa una estrategia de carga que significa "no cargar hasta que sea necesario".

Evitar la recursividad

Con la anotación **@JsonIgnoreProperties** evitamos la recursividad en la serialización indicando el campo recursivo en cada clase de datos que no deseamos mostrar en la relación, pero si incluimos estos campos de relaciones en el método toString, equals o hashCode seguiremos teniendo problemas. Para evitarlo, en las clases de datos, **excluiremos** también los campos de la relación en la generación automática de **Lombok** de los métodos **toString, equals y hashCode** incluyendo las anotaciones **@ToString** y **@EqualsAndHashCode**

Además, incorporamos la anotación para no mostrar la información que genera Hibenate al mapear sobre hibernate Lazy Initializer:
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler"})

User

```
@AllArgsConstructor         // => Constructor con todos los argumentos
@NoArgsConstructor          // => Constructor sin argumentos
@Data    // => @Getter +@Setter +@ToString +@EqualsAndHashCode +@RequiredArgsConstructor
@ToString(exclude = "role")         // Excluir toString para evitar recursividad
@EqualsAndHashCode(exclude="role") // Excluir equals + hashCode => evitar recursividad

@Entity
@Table(name = "users")
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler"})
public class User implements Serializable {
...
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "role_id", referencedColumnName = "id")
    @JsonIgnoreProperties("users")
    private Role role;
...
```

Role

```
@AllArgsConstructor         // => Constructor con todos los argumentos
@NoArgsConstructor          // => Constructor sin argumentos
@Data    // => @Getter +@Setter +@ToString +@EqualsAndHashCode +@RequiredArgsConstructor
@ToString(exclude = "users")         // Excluir toString para evitar recursividad
@EqualsAndHashCode(exclude="users") // Excluir equals y hashCode => evitar recursividad


@Entity
@Table(name = "roles")
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler"})
public class Role implements Serializable {
...
    @OneToMany(mappedBy = "role", cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @JsonIgnoreProperties("role")
    private Set<User> users = new HashSet<>();
...
```

4.2 Proyecto user-h2-jpa2-v2

user-h2-jpa2-v2

-Cambiamos variables en el **application.properties** para el nuevo proyecto

-Actualizamos en las clases de datos **campos calculados** con la anotación **@JsonProperty**.

Partiendo del proyecto user-h2-jpa2-v1 realizaremos unos pequeños cambios en las clases de datos para añadir campos calculados.

User - Modificamos

```
...
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "role_id", referencedColumnName = "id")
    @JsonIgnoreProperties("users")
    private Role role;

    // ********************************************************
    // Método getter para obtener solo el nombre del rol en minúsculas con el ID
    @JsonProperty("roleName")
    public String getRoleName() {
        String roleName=null;
        if (role != null) {
            roleName=role.getName().toLowerCase()+" ("+role.getId()+")";
        }
        return roleName;
    }

    @JsonProperty("yearNac")
    public int getYearNac() {
        return Year.now().getValue()-age;
    }
...
```

Role - Modificamos

```
...
    @OneToMany(mappedBy = "role", cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @JsonIgnoreProperties("role")
    private Set<User> users = new HashSet<>();

    // ********************************************************
    // Método para obtener el número de usuarios
    @JsonProperty("userCount")
    public int getUserCount() {
        return users != null ? users.size() : 0;
    }
...
```

Si en **NetBeans** surgen **problemas sintácticos con métodos generados por Lombok**, cambiar el nombre al campo/propiedad y luego volver al nombre inicial.

Por ejemplo, cambiamos de role a role2, grabamos y luego de nuevo de role2 a role.

4.3 Resumen

A continuación mostramos las características incluidas en cada proyecto:

	PROYECTOS	
	h2-jpa2-v1	**h2-jpa2-v2**
Thymeleaf	❌	❌
Lombok	✅	✅
Controlador	✅ **@RestController Map<String, Object>**	✅ **@RestController Map<String, Object>**
Service con Repository	✅	✅
GlobalExceptionHandler	✅	✅
Documentación HTML	✅	✅
Acceso a Datos	✅ **JpaRepository SQL**	✅ **JpaRepository SQL**
Base de Datos	✅ **H2 Database**	✅ **H2 Database**
Persistencia	✅ FILE	✅ FILE
Carga de Datos inicial	✅ **shema.sql + data.sql**	✅ **shema.sql + data.sql**
Relación entre tablas	✅ **@OneToOne @OneToMany @ManyToOne @ManyToMany @JoinColumn**	✅ **@OneToOne @OneToMany @ManyToOne @ManyToMany @JoinColumn**
Anotaciones de clase en Jackson y Lombok para Evitar recursividad	✅ **@ToString @EqualsAndHashCode @JsonIgnoreProperties**	✅ **@ToString @EqualsAndHashCode @JsonIgnoreProperties**
Anotaciones de campo en Jackson para Evitar recursividad	✅ **@JsonIgnoreProperties**	✅ **@JsonIgnoreProperties**
Añadir campos calculados a las clases JSON		✅ **@JsonProperty**