

D2 - UD5

RELACIONES 1 a N

Repositorio del proyecto:

https://github.com/AranchaC/D2_UD5.git

Descripción:

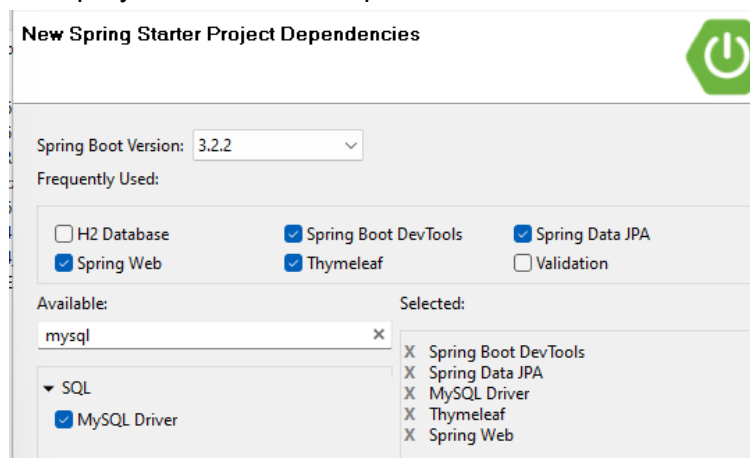
Se trata de continuar con el Quizz de la práctica anterior añadiendo la posibilidad de que un usuario tenga varias puntuaciones almacenadas en base de datos y separando en una tabla los jugadores y en otra las puntuaciones.

Al crear una nueva puntuación deberás consultar si ya existe el jugador o crear uno nuevo.

Deberás definir bien los campos de las dos tablas, claves primarias y relaciones entre ellas (clave foránea) reflejando la relación 1:N entre las entidades.

Creación / Recuperación proyecto:

Creo proyecto con estas dependencias:



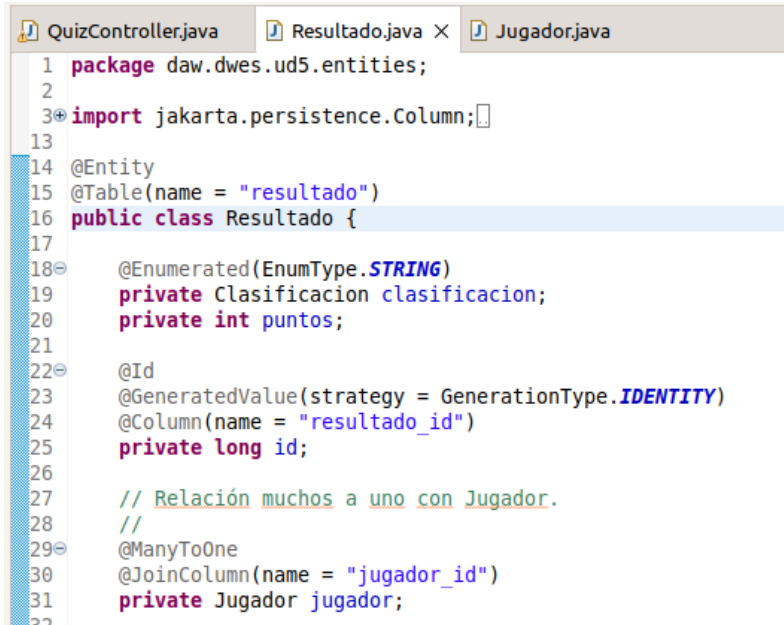
Y copio las clases, plantillas, carpeta de imágenes y application.properties del proyecto anterior.

Como ahora hay que mostrar dos tablas, jugadores y puntuaciones, tiene que haber dos entidades por lo que mi Entidad **Resultado** la voy a dividir en otra entidad **Jugador**. De esta manera la entidad Resultado se compondrá de puntos, clasificación e id, y la entidad Jugador tendrá nombre e id.

Para indicar la relación 1:N entre ambas tablas, la lógica es que un Jugador puede tener muchas puntuaciones pero un Resultado solo puede ser de un jugador. Por lo que en la tabla Resultado indicaremos la relación *muchos a uno* con la entidad Jugador, y en la clase Jugador se indicará la relación *uno a muchos* con la entidad lista Resultados.

En ambas clases he querido poner las anotaciones Table y Column para personalizar, o asegurarme, de que las tablas y columnas se mapean a MySql con los nombres indicados

Clase Entidad Resultado:



```

1 package daw.dwes.ud5.entities;
2
3 import jakarta.persistence.Column;
4
14 @Entity
15 @Table(name = "resultado")
16 public class Resultado {
17
18     @Enumerated(EnumType.STRING)
19     private Clasificacion clasificacion;
20     private int puntos;
21
22     @Id
23     @GeneratedValue(strategy = GenerationType.IDENTITY)
24     @Column(name = "resultado_id")
25     private long id;
26
27     // Relación muchos a uno con Jugador.
28     //
29     @ManyToOne
30     @JoinColumn(name = "jugador_id")
31     private Jugador jugador;
32
33 }
  
```

-Explicación código (partes nuevas):

- `@Column(name = "resultado_id")`: especifica el nombre de la columna en la tabla de la base de datos.
- `@ManyToOne`: Indica que esta entidad tiene una relación muchos a uno con la entidad Jugador.
- `@JoinColumn(name = "jugador_id")`: Especifica la columna en la tabla de la base de datos que se utiliza como clave externa para representar esta relación.
- `private Jugador jugador;`: Define un campo de tipo Jugador que representa al jugador asociado con este resultado.

Clase Entidad Jugador:

```

QuizController.java  ResultadoRepository.java  *Jugador.java  X  fir
30 import java.util.ArrayList;
14
15 @Entity
16 @Table(name = "jugador")
17 public class Jugador {
18
19     private String nombre;
20
21     @Id
22     @GeneratedValue(strategy = GenerationType.IDENTITY)
23     @Column(name = "jugador_id")
24     private long id;
25
26     // Relación uno a muchos con Resultados:
27     // Un jugador puede tener muchas puntuaciones.
28     // mappedBy para que sea bidireccional
29     @OneToMany(
30         mappedBy = "jugador",
31         cascade = CascadeType.ALL,
32         orphanRemoval = true)
33     private List<Resultado> resultados = new ArrayList<>();
34
35     public List<Resultado> getResultados() {
36         // TODO Auto-generated method stub
37         return resultados;
38     }

```

-Explicación código (partes nuevas):

- `@OneToMany(mappedBy = "jugador", cascade = CascadeType.ALL, orphanRemoval = true)`: Indica una relación uno a muchos entre la entidad *Jugador* y la entidad *Resultado*. Esto significa que un jugador puede tener muchas puntuaciones (resultados).
 - `mappedBy = "jugador"`: la relación está mapeada por el campo *jugador* en la clase *Resultado*.
 - `cascade = CascadeType.ALL`: cualquier cambio en el jugador se propagará a todas sus puntuaciones.
 - `orphanRemoval = true`: si una puntuación se elimina de la lista de puntuaciones del jugador, también se eliminará de la base de datos.
- `private List<Resultado> resultados = new ArrayList<>();`: Define una lista de objetos *Resultado* que representa las puntuaciones asociadas con este jugador.
- `public List<Resultado> getResultados() { ... }`: método getter para obtener la lista de resultados asociados con este jugador.

Y si ahora hay dos entidades, también tiene que haber 2 repositorios, uno por cada entidad.

Repositorio ResultadoRespositorio:

Con método `ultimos5resultados` que como novedad he actualizado al poner la anotación `@Query` con la que he personalizado una consulta sql ordenada por id de forma descendiente, para que salgan los últimos 5 resultados (función que ya tenía creada en la última página resultados). De esta manera se simplifica el código del controlador.

```

10 public interface ResultadoRepository extends JpaRepository<Resultado, Long>{
11
12     @Query("SELECT r FROM Resultado r ORDER BY r.id DESC LIMIT 5")
13     List<Resultado> Ultimos5Resultados();
14
15 }

```

Controlador (parte actualizada donde uso el método ultimos5resultados):

```

// Seleccionar los últimos 5 resultados (o menos si hay menos de 5)
List<Resultado> ultimosResultados = resultadoRepository.Ultimos5Resultados();

// Agregar la lista de últimos resultados al modelo
model.addAttribute("ultimosResultados", ultimosResultados);

```

Repositorio JugadorRepository (con método findByNombre):

Creo este nuevo repositorio relacionado con la entidad Jugador, donde indico método findByNombre de tipo Optional donde retorna un jugador con el nombre indicado. (método de JPA). *Más adelante se puede ver su necesidad:*

```

0 @Repository
1 public interface JugadorRepository extends JpaRepository<Jugador, Long>{
2
3     Optional<Jugador> findByNombre(String nombre);
4
5 }

```

application.properties:

Sin cambios respecto al proyecto anterior:

```

1
2 spring.datasource.url=jdbc:mysql://localhost/jugadoresQUIZZ_HP
3
4 #PARA CONECTAR EN CASA
5 #spring.datasource.url=jdbc:mysql://localhost:3307/jugadoresquizz_hp
6
7 spring.datasource.username=arancha
8 spring.datasource.password=arancha
9
10 spring.jpa.generate-ddl=true
11 spring.jpa.hibernate.ddl-auto=create
12
13 spring.jpa.show-sql=true
14 spring.jpa.properties.hibernate.format_sql=true
15
16 #show sql statement
17 logging.level.org.hibernate.SQL=debug
18
19 #show sql values
20 logging.level.org.hibernate.type.descriptor.sql=trace
21 spring.jpa.properties.hibernate.show_sql=true

```

Pruebas:

Ejecuto la api y vemos el resultado en consola, que gracias a las configuraciones en properties se puede ver en detalle las inserciones en sql.

```

Quizz_Relaciones_1aN - QuizzRelaciones1aNApplication [Spring Boot App] /home/arancha/Escritorio/sts-4.20.1.RELEASE/plugins/org.eclipse.justj.openjdk.hotsp
2024-02-08T13:55:21.178+01:00 DEBUG 32498 --- [ restartedMain] org.hibernate.SQL :
drop table if exists resultado
Hibernate:
drop table if exists resultado
2024-02-08T13:55:21.192+01:00 DEBUG 32498 --- [ restartedMain] org.hibernate.SQL :
create table jugador (
  jugador_id bigint not null auto_increment,
  nombre varchar(255),
  primary key (jugador_id)
) engine=InnoDB
Hibernate:
create table jugador (
  jugador_id bigint not null auto_increment,
  nombre varchar(255),
  primary key (jugador_id)
) engine=InnoDB
2024-02-08T13:55:21.216+01:00 DEBUG 32498 --- [ restartedMain] org.hibernate.SQL :
create table resultado (
  puntos integer not null,
  jugador_id bigint,
  resultado_id bigint not null auto_increment,
  clasificacion enum ('GRYFFINDOR','RAVENCLAW','SLYTHERIN','HUFFLEPUFF'),
  primary key (resultado_id)
) engine=InnoDB
Hibernate:
create table resultado (
  puntos integer not null,
  jugador_id bigint,
  resultado_id bigint not null auto_increment,
  clasificacion enum ('GRYFFINDOR','RAVENCLAW','SLYTHERIN','HUFFLEPUFF'),
  primary key (resultado_id)
) engine=InnoDB
2024-02-08T13:55:21.240+01:00 DEBUG 32498 --- [ restartedMain] org.hibernate.SQL :
alter table resultado
add constraint FKbs3ur9yttaclujpd99gpogild
foreign key (jugador_id)
references jugador (jugador_id)
Hibernate:
alter table resultado
add constraint FKbs3ur9yttaclujpd99gpogild
foreign key (jugador_id)
references jugador (jugador_id)
2024-02-08T13:55:21.289+01:00 INFO 32498 --- [ restartedMain] j.LocalContainerEntityManagerFactoryBean : Initialized JF
2024-02-08T13:55:21.348+01:00 WARN 32498 --- [ restartedMain] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.o

```

Accedo a mysql con el usuario arancha y compruebo que se han creado las tablas con los tipo de datos correspondientes:

```

arancha@daw2-01:~$ mysql -u arancha -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 175
Server version: 8.0.36-0ubuntu0.22.04.1 (Ubuntu)

Copyright (c) 2000, 2024, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+-----+
| Database |
+-----+
| harryPotterBD |
| information schema |
| jugadoresQUIZZ HP |
| performance_schema |
+-----+
4 rows in set (0,01 sec)

mysql> use jugadoresQUIZZ_HP;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;

```

```

mysql> show tables;
+-----+
| Tables_in_jugadoresQUIZZ_HP |
+-----+
| jugador |
| resultado |
+-----+
2 rows in set (0,01 sec)

mysql> describe jugador;
+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+
| jugador_id | bigint | NO | PRI | NULL | auto_increment |
| nombre | varchar(255) | YES | | NULL | |
+-----+
2 rows in set (0,00 sec)

mysql> describe resultado;
+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+
| puntos | int | NO | MUL | NULL | |
| jugador_id | bigint | YES | MUL | NULL | |
| resultado_id | bigint | NO | PRI | NULL | auto_increment |
| clasificacion | enum('GRYFFINDOR', 'RAVENCLAW', 'SLYTHERIN', 'HUFFLEPUFF') | YES | | NULL | |
+-----+
4 rows in set (0,00 sec)

mysql>

```

Pruebo en el navegador y funciona sin ningún error, los datos se vuelcan en MySQL:



```
mysql> select * from jugador;
+-----+-----+
| jugador_id | nombre |
+-----+-----+
|          1 | aranchaaa |
+-----+-----+
1 row in set (0,00 sec)

mysql> select * from resultado;
+-----+-----+-----+-----+
| puntos | jugador_id | resultado_id | clasificacion |
+-----+-----+-----+-----+
|      18 |          1 |             1 | RAVENCLAW |
+-----+-----+-----+-----+
1 row in set (0,00 sec)

mysql>
```

Volver a empezar

Últimos 5 Resultados

Nombre	Puntos	Clasificación
eva	16	RAVENCLAW
eva	23	GRYFFINDOR

```
MariaDB [jugadoresquizz_hp]> select * from jugador;
+-----+-----+
| jugador_id | nombre |
+-----+-----+
|          1 | eva |
+-----+-----+
1 row in set (0.001 sec)

MariaDB [jugadoresquizz_hp]> select * from resultado;
+-----+-----+-----+-----+
| puntos | jugador_id | resultado_id | clasificacion |
+-----+-----+-----+-----+
|      23 |          1 |             1 | GRYFFINDOR |
|      16 |          1 |             2 | RAVENCLAW |
+-----+-----+-----+-----+
2 rows in set (0.001 sec)

MariaDB [jugadoresquizz_hp]>
```


CAPAS DE APLICACIÓN EN PAQUETES SEPARADOS

Para hacer la api más funcional, voy a separar el controlador en dos controladores, y en servicios. De esta manera habrá un **quizzController** para la lógica del juego, un **resultadoController** para las funciones de consultas sql (los métodos CRUD), un servicio **clasificaciónService** para el método calcularClasificacion y otro servicio **ResultadoService** para el método obtenerResultado.

Servicio ClasificacionService:

Simplemente traslado ahí el método calcularClasificación del controlador, sin ningún cambio:

```

7  @Service
8  public class ClasificacionService {
9
10     public Clasificacion calcularClasificacion(int puntos) {
11         // determinar la clasificación según los puntos
12         if (puntos >= 20) {
13             return Clasificacion.GRYFFINDOR;
14         } else if (puntos >= 15) {
15             return Clasificacion.RAVENCLAW;
16         } else if (puntos >= 10) {
17             return Clasificacion.SLYTHERIN;
18         } else {
19             return Clasificacion.HUFFLEPUFF;
20         }
21     } //calcularClasif

```

Servicio ResultadoService:

Y en este servicio traslado el método obtenerResultado del controlador:

```

8  @Service
9  public class ResultadoService {
10
11     public Resultado obtenerResultado(HttpSession session) {
12         Resultado resultado = (Resultado) session.getAttribute("resultado");
13         // Si no existe en la sesión, crear uno nuevo
14         //y se guarda en la sesión:
15         if (resultado == null) {
16             resultado = new Resultado();
17             session.setAttribute("resultado", resultado);
18         } //if
19         return resultado;
20     } //obtenerResultado
21
22 } //Service
--

```

Controlador ResultadoController (consultas CRUD):

En este controlador como solo uso el repositorio de resultado, creo variable resultadoRepository para poder acceder a sus valores, lo demás es igual que el código que estaba en el otro controlador pero que hemos trasladado, todos los métodos de consultas CRUD (get, post, delete, put).

Importante, incluir la anotación **@RestController**:


```

19 @RestController
20 public class ResultadoController {
21
22     private final ResultadoRepository resultadoRepository;
23
24     @Autowired
25     public ResultadoController(ResultadoRepository resultadoRepository) {
26         this.resultadoRepository = resultadoRepository;
27     }
28
29     // MÉTODOS CONSULTAS SQL //
30
31     @GetMapping("/resultados")
32     public List<Resultado> obtenerTodosLosResultados() {
33         return resultadoRepository.findAll();
34     } // Obtener Todos
35
36     @GetMapping("/resultados/{id}")
37     public Optional<Resultado> buscarResultadoId(@PathVariable ("id") Long id) {

```

Controlador QuizController:

Y en el controlador principal, como usaremos los dos repositorios además de los dos servicios (porque aquí usamos los dos métodos que hemos separado en servicios) tenemos que hacer llamada a estas cuatro clases, además de incluirlas en el controlador para inicializarlas:

```

@Controller
public class QuizController {

    private final ResultadoRepository resultadoRepository;
    private final JugadorRepository jugadorRepository;

    private final ResultadoService resultadoService;
    private final ClasificacionService clasificacionService;

    @Autowired
    public QuizController(
        ResultadoRepository resultadoRepository,
        JugadorRepository jugadorRepository,
        ResultadoService resultadoService,
        ClasificacionService clasificacionService) {
        this.resultadoRepository = resultadoRepository;
        this.jugadorRepository = jugadorRepository;
        this.resultadoService = resultadoService;
        this.clasificacionService = clasificacionService;
    }

```

Y ahora, para llamar a la función **obtenerResultado**, tenemos que poner delante el nombre de la clase servicio:

```

// Obtener el objeto Resultado de la sesión
Resultado resultado = resultadoService.obtenerResultado(session);

```

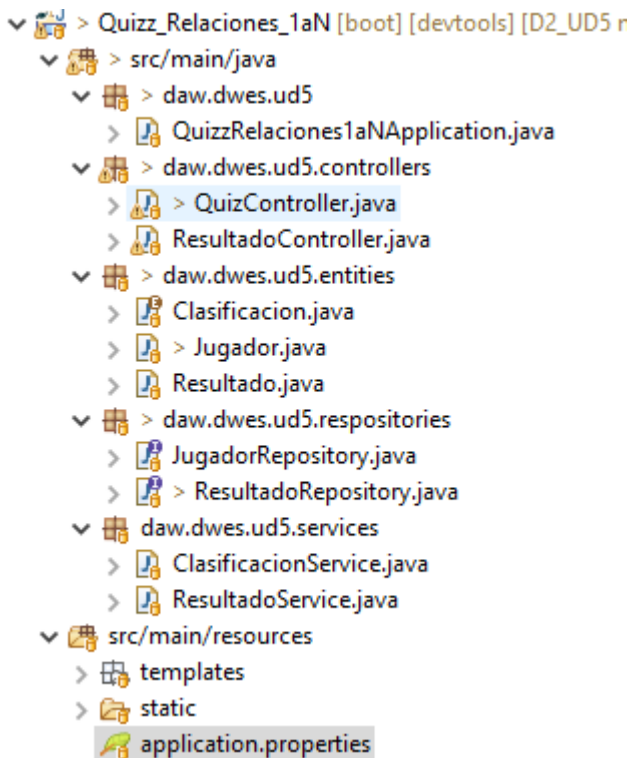
Lo mismo pasa para llamar a la función **calcularClasificacion** del servicio clasificación:

```

// y actualizamos la clasificación, accediendo a los puntos:
resultado.setClasificacion(clasificacionService.calcularClasificacion
    (resultado.getPuntos()));

```

Y por lo tanto el proyecto, con la separación de las cuatro capas quedaría de la siguiente manera:



AÑADIR CAMPO FECHA PARA REGISTRAR CADA PUNTUACIÓN:

-Entidad Resultado:

Para añadir un campo fecha para ver la fecha de la puntuación en la tabla de resultados, hay que añadir una propiedad Date a la entidad Resultado:

```
@Temporal(TemporalType.TIMESTAMP) // Especifica el tipo temporal como TIMESTAMP
@Column(name = "fecha")
private Date fecha;
```

La anotación **@Temporal** se utiliza en JPA para indicar cómo se debe mapear una propiedad de tipo Date o Calendar a la base de datos. La anotación se utiliza con **@TemporalType** para especificar la precisión de la información temporal que se almacenará en la base de datos.

- TemporalType.DATE: Mapea la propiedad a un tipo de fecha de base de datos.
- TemporalType.TIME: Mapea la propiedad a un tipo de tiempo de base de datos.
- **TemporalType.TIMESTAMP**: (elegida) Mapea la propiedad a un tipo de fecha y hora de base de datos con la precisión de un timestamp.

-Controlador QuizController:

Y en el controlador, en el método **paginaNombre** que es la página donde se gestiona y almacena la puntuación, y por lo tanto se crea el objeto Resultado, mediante la función **.setFecha** le asignamos al resultado la fecha actual con **new Date()**:

```

@PostMapping("/paginaNombre")
public String paginaNombre(
    @RequestParam(name = "nombre") String nombre,
    HttpSession session,
    Model model) {

    // Obtener el objeto Resultado de la sesión
    Resultado resultado = resultadoService.obtenerResultado(session);

    // Establecer la fecha actual
    resultado.setFecha(new Date());
}

```

-Plantilla/Vista finalResultado:

Para mostrar la fecha en la tabla que ya tenemos creada, bastaría con crear una nueva columna para llamar al valor fecha con **resultado.fecha**:

```

31 </form>
32
33 <table border="1">
34   <thead>
35     <tr>
36       <th>Nombre</th>
37       <th>Puntos</th>
38       <th>Clasificación</th>
39       <th>Hora y fecha</th>
40     </tr>
41   </thead>
42   <tbody>
43     <!-- For each sobre la lista de resultados -->
44     <tr th:each="resultado : ${ultimosResultados}">
45       <td th:text="${resultado.jugador.nombre}" />
46       <td th:text="${resultado.puntos}" />
47       <td th:text="${resultado.clasificacion}" />
48       <td th:text="${resultado.fecha}" />

```

-Prueba:

Ejecuto api y veo el resultado en el navegador y en mysql:

The screenshot shows two parts: a MySQL terminal window at the top and a web application interface at the bottom.

MySQL Terminal: The terminal shows a query result for the 'jugadoresquizz_hp' table. The output is as follows:

puntos	fecha	jugador_id	resultado_id	clasificacion
23	2024-02-10 20:12:34.000000	1	1	GRYFFINDOR

The terminal also shows the command: `MariaDB [jugadoresquizz_hp]> |`

Web Application Interface: The interface shows a search bar labeled "Filtrar por jugador:" with a "Filtrar" button. Below the search bar is a table with the following data:

Nombre	Puntos	Clasificación	Hora y fecha
aran	23	GRYFFINDOR	Sat Feb 10 20:12:34 CET 2024

Pero para darle un formato más legible y visual, en la misma plantilla formateo la fecha con **dates.format**, método que sacado de los siguientes links:

<https://springhow.com/handling-date-objects-in-thymeleaf/>

<https://www.youtube.com/watch?v=yyJBO1jlk18>

<https://stackoverflow.com/questions/39860643/formatting-date-in-thymeleaf>

El primer link indica que se puede crear un campo fecha en la plantilla sin necesidad de crear un atributo date en el código. Eso puede ser útil cuando solo nos interese la fecha de forma visual, es decir, solo verla en el navegador y que no nos interese como atributo de nuestro objeto. En este caso no lo voy a usar porque también quiero que la fecha quede guardada en la tabla de la bbdd como un atributo más de mi objeto entidad.

```
<!-- For each sobre la lista de resultados -->
<tr th:each="resultado : ${ultimosResultados}">
  <td th:text="${resultado.jugador.nombre}" />
  <td th:text="${resultado.puntos}" />
  <td th:text="${resultado.clasificacion}" />
  <td th:text="${#dates.format(resultado.fecha, 'HH:mm a dd-MMM')}" />
</tr>
```

Y se mostraría de la siguiente manera:

Nombre	Puntos	Clasificación	Hora y fecha
juan	24	GRYFFINDOR	20:33 p. m. 10-feb
aran	26	GRYFFINDOR	20:21 p. m. 10-feb
pepita	15	RAVENCLAW	20:19 p. m. 10-feb

CONDICIÓN SI EXISTE JUGADOR

Si existe se asocia la puntuación a ese jugador, y si no existe, se crea nuevo jugador y se le asocia la puntuación.

En **QuizControler**, en el método **paginaNombre** que es dónde se gestionan los resultados, creo condición de si existe el jugador. Para ello creo variable **jugadorOptional** de tipo **Optional**, que con el método **findByNombre** que tenemos en el **jugadorRepository** busca si hay un jugador con el nombre que pasamos por parámetro, y le asignamos el valor de ese jugador.

```
// Verificar si el jugador ya existe en la base de datos
Optional<Jugador> jugadorOptional = jugadorRepository.findByNombre(nombre);

if (jugadorOptional.isPresent()) {
    // Si el jugador ya existe, asociar la puntuación al jugador existente:
    Jugador jugadorExistente = jugadorOptional.get();
    resultado.setJugador(jugadorExistente);
} else {
    // Si el jugador no existe, crear un nuevo jugador y asociar la puntuación
    Jugador nuevoJugador = new Jugador();
    nuevoJugador.setNombre(nombre);
    nuevoJugador.getPuntuaciones().add(resultado);
    jugadorRepository.save(nuevoJugador);
    resultado.setJugador(nuevoJugador);
} //if-Else

// Guardar el resultado en el repositorio con .save:
resultadoRepository.save(resultado);
```

Y con if/else gestiono la lógica de la siguiente manera:

- **if (jugadorOptional.isPresent()) { ... }:** se verifica si este jugadorOptional existe en la bbdd.
 - **Jugador jugadorExistente = jugadorOptional.get();** Si existe, se le asigna este jugadorOptional encontrado a la variable jugadorExistente con get().
 - **resultado.setJugador(jugadorExistente);** se asigna el jugadorExistente al jugador de la bbdd con setJugador().
- **else { ... }:** Si no se encuentra un jugador con el nombre:
 - **Jugador nuevoJugador = new Jugador();** Se crea un nuevo objeto Jugador.
 - **nuevoJugador.setNombre(nombre);** se asigna el nombre proporcionado como el nombre del nuevo jugador.
 - **nuevoJugador.getPuntuaciones().add(resultado);** Se agrega el resultado actual a la lista de puntuaciones del nuevo jugador.
 - **jugadorRepository.save(nuevoJugador);** Se guarda el nuevo jugador en la base de datos con el repositorio de jugadores.
 - **resultado.setJugador(nuevoJugador);** se asigna el nuevo jugador como el jugador asociado al resultado de la bbdd.

Prueba:

Para hacer la prueba, voy a crear más resultados con el mismo usuario, con aran:

Nombre	Puntos	Clasificación	Hora y fecha
aran	17	RAVENCLAW	00:10 a. m. 11-feb
aran	8	HUFFLEPUFF	00:09 a. m. 11-feb
pavel	18	RAVENCLAW	20:55 p. m. 10-feb
juan	24	GRYFFINDOR	20:33 p. m. 10-feb
aran	26	GRYFFINDOR	20:21 p. m. 10-feb

Se puede ver que en la tabla resultado de MySql(ver abajo) hay 3 valores con el mismo id, id 2, por lo que un jugador se ha repetido ya que tiene diferentes puntuaciones/resultados, pero en la tabla de jugador vemos que solo hay un jugador aran con id 2, por lo tanto el código funciona correctamente, ya que el jugador no se duplica pero si puede tener varios resultados. **Aquí podemos ver claramente la lógica de OneToMany / ManyToOne.**

```
MariaDB [jugadoresquizz_hp]> select * from resultado;
```

puntos	fecha	jugador_id	resultado_id	clasificacion
15	2024-02-10 20:19:41.000000	1	1	RAVENCLAW
26	2024-02-10 20:21:21.000000	2	2	GRYFFINDOR
24	2024-02-10 20:33:49.000000	3	3	GRYFFINDOR
18	2024-02-10 20:55:24.000000	4	4	RAVENCLAW
8	2024-02-11 00:09:23.000000	2	5	HUFFLEPUFF
17	2024-02-11 00:10:09.000000	2	6	RAVENCLAW

6 rows in set (0.001 sec)

```
MariaDB [jugadoresquizz_hp]> select * from jugador;
```

jugador_id	nombre
1	pepita
2	aran
3	juan
4	pavel

Es curioso ver también el resultado en consola:

```
2024-02-11T00:21:07.776+01:00 DEBUG 6392 --- [nio-8080-exec-4] org.hibernate.SQL
    select
      r1_0.resultado_id,
      r1_0.clasificacion,
      r1_0.fecha,
      r1_0.jugador_id,
      r1_0.puntos
    from
      resultado r1_0
    left join
      jugador j1_0
      on j1_0.jugador_id=r1_0.jugador_id
    where
      j1_0.nombre=?
Hibernate:
    select
      r1_0.resultado_id,
      r1_0.clasificacion,
      r1_0.fecha,
      r1_0.jugador_id,
      r1_0.puntos
    from
      resultado r1_0
    left join
      jugador j1_0
      on j1_0.jugador_id=r1_0.jugador_id
    where
      j1_0.nombre=?
2024-02-11T00:21:07.780+01:00 DEBUG 6392 --- [nio-8080-exec-4] org.hibernate.SQL
```

FUNCIONALIDAD FILTRAR RESULTADOS POR NOMBRE DE JUGADOR

Para añadir esta funcionalidad, primero he añadido en la plantilla correspondiente el campo de texto y el botón, en la plantilla de **finalResultados**, y luego he aplicado la lógica en el controlador.

La lógica que he seguido es, como el campo de texto para introducir el nombre a buscar lo he puesto en la última página, en la de resultados, pues he *añadido una página más* para mostrar ahí los resultados filtrados, es decir, el formulario de campo de texto para filtrar los resultados por nombre redirigirá a otra página, gracias a la función del controlador, donde mostrará los resultados filtrados.

Plantilla finalResultado.html:

El action va dirigido a sí mismo, pues es su método del controlador el que gestiona esta función y devuelve la siguiente página:

```
6 <!-- Agregar un formulario para filtrar por jugador -->
7 <form th:action="@{/finalResultado}" method="post">
8   <label for="nombre" >Ver resultados por jugador:</label>
9   <input type="text" id="nombre" name="nombre" placeholder="Pon el nombre">
0   <button type="submit" class="botonF">Filtrar</button>
1 </form>
2
```

ResultadoRepository:

Para filtrar los resultados creo un método usando usar `findBy`, método de JPA, y como queremos *buscar* el nombre del jugador, llamaremos al método `findByJugadorNombre`, y recibe por parámetro el nombre:

```
public interface ResultadoRepository extends JpaRepository<Resultado, Long>{

    @Query("SELECT r FROM Resultado r ORDER BY r.fecha DESC LIMIT 5")
    List<Resultado> Ultimos5Resultados();

    List<Resultado> findByJugadorNombre(String nombre);
}
```

Controlador QuizController:

```
@PostMapping("/finalResultado")
public String finalResultado(
    @RequestParam(name = "nombre")String nombre,
    Model model) {
    // Buscar resultados por nombre de jugador
    List<Resultado> resultadosFiltrados =
        resultadoRepository.findByJugadorNombre(nombre);

    //comentar error de hacerlo con optional:

    if (resultadosFiltrados.isEmpty()) {
        // Si no se encuentran resultados, mostrar un mensaje de alerta
        model.addAttribute("mensaje", "No se encontraron resultados para el jugador " + nombre);
    } else {
        // Si se encuentran resultados, agregarlos al modelo
        model.addAttribute("resultadosFiltrados", resultadosFiltrados);
    }

    return "finalResultadoFiltrado";
}
```


Para hacer este método me he basado en los ejemplos del resto de métodos de la api, pero procedo a explicar:

- **@PostMapping("/finalResultado")**: indica que este método maneja las solicitudes POST que llegan a la URL "/finalResultado".
- **Parámetros**:
 - **@RequestParam(name = "nombre")String nombre**: Este parámetro obtiene el valor del parámetro "nombre" enviado en la solicitud POST.
 - **Model model**: El objeto Model se utiliza para pasar datos al modelo
- **List<Resultado>resultadosFiltrados=resultadoRepository.findByJugadorNombre(nombre)**: busca resultados en el resultadoRepository filtrados por el nombre del jugador especificado y los asigna a la lista resultadosFiltrados.
- **if (resultadosFiltrados.isEmpty())**: verifica si la lista de resultados filtrados está vacía, si no hay resultados.
- **model.addAttribute("mensaje", "No se _____")**: Si no hay resultados, se agrega un mensaje de alerta al modelo para mostrar en la vista.
- **model.addAttribute("resultadosFiltrados", resultadosFiltrados)**: Si se encuentran resultados para el jugador, se agregan al modelo para mostrar en la vista.
- **return "finalResultadoFiltrado"**: Finalmente, se devuelve el nombre de la vista que se debe mostrar. En este caso, la vista "finalResultadoFiltrado".

Plantilla/vista finalResultadoFiltrado.html:

Y por último creo la plantilla correspondiente donde se mostrará una tabla con los resultados filtrados por nombre del jugador. He seguido el ejemplo de la plantilla anterior:

```

10 <h2>Resultados del Jugador <span class="nombre" th:text="${param.nombre}"></span></h2>
11
12 <!-- Verificar si hay un mensaje -->
13 <div th:if="${mensaje != null}">
14   <p th:text="${mensaje}" />
15 </div>
16

```

Añado una línea donde aparece una frase con el nombre del jugador, que como este nombre lo recibe por parámetro, para obtener ese nombre hay que poner param.

Luego, siguiendo la lógica del código del controlador, con **th:if** se dice si el mensaje no es null, es decir si existe un mensaje de error y por lo tanto la lista de resultadosFiltrados está vacía, se muestra el mensaje de error.

```

<div class="tabla" th:if="${mensaje == null}">
  <table border="1">
    <thead>
      <tr>
        <th>Puntos</th>
        <th>Clasificación</th>
        <th>Hora y Fecha</th>
      </tr>
    </thead>
    <tbody>
      <!-- For each sobre la lista de resultadosFiltrados -->
      <tr th:each="resultado : ${resultadosFiltrados}">
        <td th:text="${resultado.puntos}" />
        <td th:text="${resultado.clasificacion}" />
        <td th:text="${#dates.format(resultado.fecha, 'HH:mm a dd-MMM')}" />
      </tr>
    </tbody>
  </table>
</div>

```

Y si el mensaje es null, si no hay mensaje de error y por lo tanto hay lista de resultadosFiltrados, se muestra la tabla con los resultados. Que se muestran, siguiendo el ejemplo de la plantilla ya realizada en finalResultado, con un for each, que en thymeleaf se usa con **th:each**, y en este caso indicando la lista a la que queremos acceder que es resultadosFiltrados.

```

<br>
<a th:href="@{/}"><button class="botonIniciar">Volver a empezar</button></a>
<form th:action="@{/finalResultado}" method="post">
  <label for="nombre" >Ver resultados de otro jugador:</label>
  <input type="text" name="nombre" placeholder="Nombre del jugador">
  <button type="submit" class="botonF">Filtrar</button>
</form>

```

Y como punto adicional, en esta misma plantilla vuelvo a añadir un formulario para ahí permitir que se siga filtrando los resultados para otro jugador, de esta manera podemos seguir filtrando resultados por nombre desde la misma página, sin necesidad de volver a hacer todo el quiz desde el inicio.

Otro adicional es que he añadido un botón para volver a empezar el quiz, que redirige a la página inicial, la página raíz donde se inicia el quiz. Funcionalidad también añadida en la página anterior.

Prueba filtrar:

Para hacer la prueba, voy a seguir con los valores anteriores. Como hay varios resultados para el usuario aran, voy a filtrar el usuario aran para que solo se vean sus resultados:

Volver a empezar

Filtrar por jugador:

Nombre	Puntos	Clasificación	Hora y fecha
aran	17	RAVENCLAW	00:10 a. m. 11-feb
aran	8	HUFFLEPUFF	00:09 a. m. 11-feb
pavel	18	RAVENCLAW	20:55 p. m. 10-feb
juan	24	GRYFFINDOR	20:33 p. m. 10-feb
aran	26	GRYFFINDOR	20:21 p. m. 10-feb

Y se muestran correctamente todos los resultados del jugador aran:

Resultados del Jugador ARAN

Puntos	Clasificación	Hora y Fecha
26	GRYFFINDOR	20:21 10-02
8	HUFFLEPUFF	00:09 11-02
17	RAVENCLAW	00:10 11-02

Volver a empezar

Ver resultados de otro jugador:

Vemos también el resultado en consola:

```
2024-02-11T00:21:07.780+01:00 DEBUG 6392 --- [nio-8080-exec-4] org.hibernate.SQL
    select
      j1_0.jugador_id,
      j1_0.nombre
    from
      jugador j1_0
    where
      j1_0.jugador_id=?
Hibernate:
    select
      j1_0.jugador_id,
      j1_0.nombre
    from
      jugador j1_0
    where
      j1_0.jugador_id=?
```

Pruebo a filtrar otro nombre, pavel con dos resultados:



Volver a empezar

Filtrar por jugador:

Nombre	Puntos	Clasificación	Hora y fecha
pavel	19	RAVENCLAW	00:27 a. m. 11-feb
aran	17	RAVENCLAW	00:10 a. m. 11-feb
aran	8	HUFFLEPUFF	00:09 a. m. 11-feb
pavel	18	RAVENCLAW	20:55 p. m. 10-feb
juan	24	GRYFFINDOR	20:33 p. m. 10-feb



Resultados del Jugador **PAVEL**

Puntos	Clasificación	Hora y Fecha
18	RAVENCLAW	20:55 10-02
19	RAVENCLAW	00:27 11-02

Volver a empezar

Ver resultados de otro jugador:

Filtrar en orden descendiente puntuación (primero las puntuaciones mayores):

Para ello se puede hacer personalizando una query en el método de `findByJugadorNombre` del `resultadoRepository` pero para aprovechar los métodos intrínsecos de JPA, podemos conseguirlo añadiendo `OrderBy` y `Desc`, como lo queremos ordenar por puntos hay que añadir `OrderByPuntosDesc`:

```
public interface ResultadoRepository extends JpaRepository<Resultado, Long>{
    @Query("SELECT r FROM Resultado r ORDER BY r.fecha DESC LIMIT 5")
    List<Resultado> Ultimos5Resultados();

    List<Resultado> findByJugadorNombreOrderByPuntosDesc(String nombre);
}
```

Y en el controlador, ahora llamo a este método:

```
@PostMapping("/finalResultado")
public String finalResultado(
    @RequestParam(name = "nombre")String nombre,
    Model model) {
    // Buscar resultados por nombre de jugador
    List<Resultado> resultadosFiltrados =
        resultadoRepository.findByJugadorNombreOrderByPuntosDesc(nombre);
```

Estos métodos se pueden encontrar en este enlace de la documentación oficial:
<https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html>

Pruebo con estos datos que son 5 resultados del jugador aran:

Ver resultados por jugador:

Nombre	Puntos	Clasificación	Hora y fecha
aran	12	SLYTHERIN	20:34 p. m. 11-feb
aran	34	GRYFFINDOR	20:33 p. m. 11-feb
aran	26	GRYFFINDOR	20:33 p. m. 11-feb
aran	15	RAVENCLAW	20:32 p. m. 11-feb
aran	7	HUFFLEPUFF	20:32 p. m. 11-feb

Y al filtrar ese nombre, se puede ver que ahora los resultados se muestran ordenados según los puntos de mayor a menor.

Resultados del Jugador ARAN

Puntos	Clasificación	Hora y Fecha
34	GRYFFINDOR	20:33 p. m. 11-feb
26	GRYFFINDOR	20:33 p. m. 11-feb
15	RAVENCLAW	20:32 p. m. 11-feb
12	SLYTHERIN	20:34 p. m. 11-feb
7	HUFFLEPUFF	20:32 p. m. 11-feb

Ver resultados de otro jugador:

IMPLEMENTAR LISTAR 5 PUNTUACIONES POR JUGADOR COMO MÁXIMO Y QUE SEAN LAS 5 MÁS ALTAS

Para implementar esta funcionalidad es parecido al apartado anterior. Aprovecho un método de consulta de JPA. Usamos el mismo método que ya está creado **findByJugadorNombreOrderByPuntosDesc(String nombre)** en el resultadoRepository, y

en este caso para que se muestren 5 resultados, hay que añadir al método First5 o Top5, y para que sean las puntuaciones más altas se haría con el OrderBy y Desc que ya estaba aplicado:

```

3 public interface ResultadoRepository extends JpaRepository<Resultado, Long>{
4
5     @Query("SELECT r FROM Resultado r ORDER BY r.fecha DESC LIMIT 5")
6     List<Resultado> Ultimos5Resultados();
7
8     List<Resultado> findTop5ByJugadorNombreOrderByPuntosDesc(String nombre);
9 }

```

Siguiendo la lógica de sql, y como en JPA ha funcionado el orderBy y Desc, se me ocurrió poner un Limit5 al método, pero dió error en la consola de que no existe. Por lo que en el link siguiente encontré el método de consulta Top y First para limitar la búsqueda:

<https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html>

Y por lo tanto en el controlador vuelvo a cambiar la llamada al método de resultados filtrados por este nuevo método modificado:

```

@PostMapping("/finalResultado")
public String finalResultado(
    @RequestParam(name = "nombre")String nombre,
    Model model) {
    // Buscar resultados por nombre de jugador
    List<Resultado> resultadosFiltrados =
        resultadoRepository.findByJugadorNombreOrderByPuntosDesc(nombre);
}

```

Prueba:

Pruebo con estos datos, que vemos que son 7 resultados del mismo id, id 1 jugador aran:

MariaDB [jugadoresquizz_hp]> select * from resultado;

puntos	fecha	jugador_id	resultado_id	clasificacion
7	2024-02-11 20:49:11.000000	1	1	HUFFLEPUFF
16	2024-02-11 20:49:48.000000	1	2	RAVENCLAW
35	2024-02-11 20:50:27.000000	1	3	GRYFFINDOR
13	2024-02-11 20:51:09.000000	1	4	SLYHERIN
12	2024-02-11 20:51:59.000000	1	5	SLYHERIN
18	2024-02-11 20:52:46.000000	1	6	RAVENCLAW
26	2024-02-11 20:53:46.000000	1	7	GRYFFINDOR

7 rows in set (0.000 sec)

MariaDB [jugadoresquizz_hp]>

Ver resultados por jugador:

Nombre	Puntos	Clasificación	Hora y fecha
aran	26	GRYFFINDOR	20:53 p. m. 11-feb
aran	18	RAVENCLAW	20:52 p. m. 11-feb
aran	12	SLYHERIN	20:51 p. m. 11-feb

Y ahora filtro por su nombre, y funciona, de los 7 resultados solo se muestran 5 y son los más altos gracias al método que ya teníamos creado:

Resultados del Jugador ARAN		
Puntos	Clasificación	Hora y Fecha
35	GRYFFINDOR	20:50 p. m. 11-feb
26	GRYFFINDOR	20:53 p. m. 11-feb
18	RAVENCLAW	20:52 p. m. 11-feb
16	RAVENCLAW	20:49 p. m. 11-feb
13	SLYTHERIN	20:51 p. m. 11-feb

Volver a empezar

Ver resultados de otro jugador:

COMENTARIOS

Repositorio GIT: https://github.com/AranchaC/D2_UD5.git

A lo largo de la realización de esta práctica he ido teniendo algunos problemas sin mucha importancia, como querer limitar la lista de filtrados al poner limit cuando no existe en JPA, que al ver la documentación vi que se usaba First o Top. También tuve un pequeño problema al acceder desde la plantilla a la propiedad nombre, ya que ahora se ha separado en dos entidades y en la plantilla se accede a los valores de resultado cuando ahora nombre es de jugador, y usando ChatGPT me dio el detalle de que para acceder ahora al valor de jugador (cuando estamos haciendo llamada recorriendo Resultado) se hace así: resultado.jugador.nombre:

```
<!-- For each sobre la lista de resultados -->
<tr th:each="resultado : ${ultimosResultados}">
  <td th:text="${resultado.jugador.nombre}" />
  <td th:text="${resultado.puntos}" />
  <td th:text="${resultado.clasificacion}" />
  <td th:text="${#dates.format(resultado.fecha, 'HH:mm a dd-MMM')}" />
</tr>
```

Para añadir la fecha, ha sido fácil, pero me ha tocado refrescar conocimientos sobre como usar el valor Date. Por otro lado me ha gustado mucho como personalizar la vista de este campo de manera que se pueda mostrar de la forma que más me gusta.

En general no ha sido una práctica difícil, me ha gustado realizarla, ver cómo se mapean los datos entre sts y mysql de manera casi automática. También usar y ver los métodos de consulta que incluye JPA y que hace muy sencillo el código. Las cosas que no he sabido como hacer inicialmente me he servido de ayuda de varios documentos del temario y de la documentación oficial de spring boot. Y me ha parecido muy interesante la unión de las tablas con OneToMany / ManyToOne.