IES EL RINCÓN

2º Desarrollo de Aplicaciones Multiplataforma

2024 - 2025

Axon - Documentación

Proyecto de Trimestre

Presenta:

Ancor García Guedes

Jorge Gabriel Arcalá Gonzalez

Las Palmas de Gran Canaria, Mayo 2025

ancorgarcia63@gmail.com

jg.arcalagonzalez@gmail.com

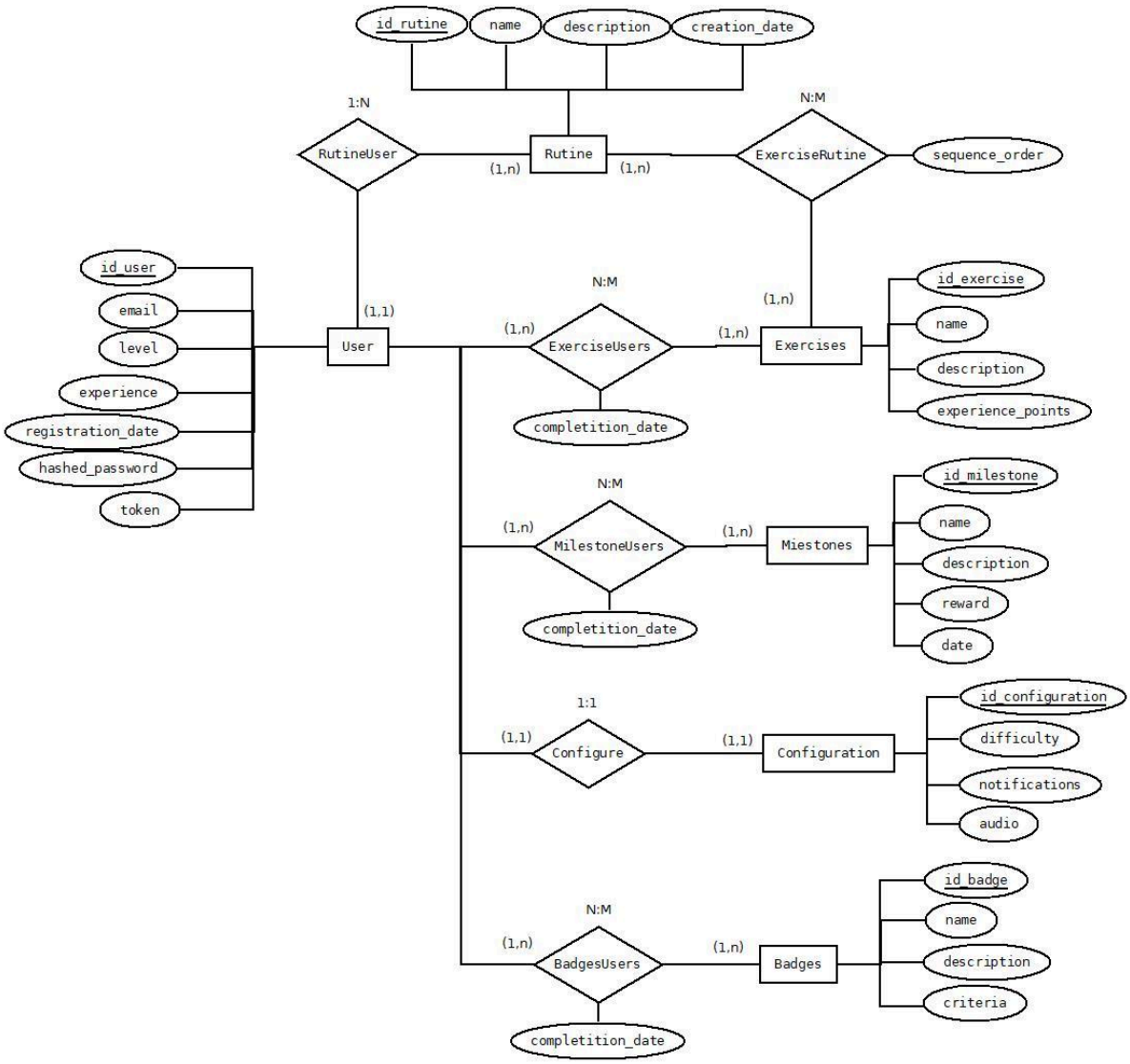
GitHub del FrontEnd

GitHub del BackEnd

Índice

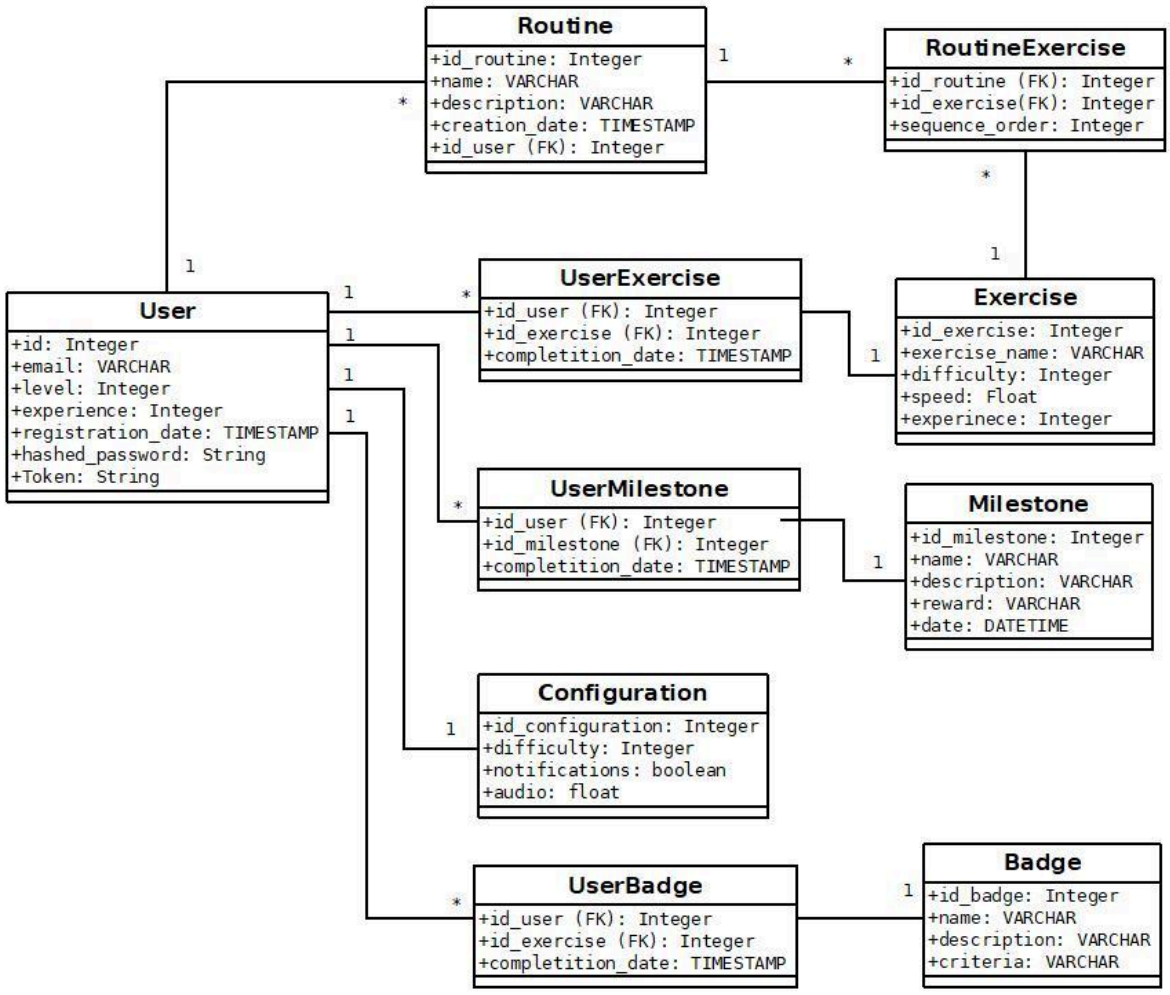
Modelo E/R -

Desde el punto de vista del modelo E/R, el sistema consiste en varias entidades clave: **Exercise**, **Routine**, **Users**, **Milestones**, **Badges**, y **Configuration**. Las relaciones entre ellas se representan con vínculos como **RoutineExercise**, que conecta rutinas y ejercicios, y **UserExercise**, que asocia a los usuarios con ejercicios completados. Además, existen asociaciones entre **Users** y **Milestones** a través de la entidad **UserMilestone**, así como entre **Users** y **Badges** mediante la relación **UserBadge**. Cada entidad se representa como un rectángulo con atributos dentro, y las relaciones como líneas que conectan las entidades.



Modelo UML -

En el modelo UML, las entidades se representarían como clases con sus atributos listados dentro de ellas. Por ejemplo, la clase **Exercise** tiene atributos como *id_exercise*, *exercise_name*, *difficulty*, *speed*, y *experience*. Las relaciones entre las clases serían representadas con asociaciones, como la relación **RoutineExercise** que une **Exercise** y **Routine**. Las asociaciones pueden ser con multiplicidad, indicando, por ejemplo, que un **User** puede tener muchos **Milestones** a través de la clase de relación **UserMilestone**.



Modelo Relacional -

En el modelo relacional, los datos se estructuran en tablas. Cada entidad se representa como una tabla con columnas para sus atributos. Por ejemplo, la tabla **Exercise** tendrá columnas como *id_exercise*, *exercise_name*, *difficulty*, *speed*, y *experience*. Las relaciones entre entidades se representan mediante claves foráneas: en la tabla **RoutineExercise**, *id_routine* y *id_exercise* son claves foráneas que vinculan las rutinas y los ejercicios.

Las tablas **UserExercise**, **UserMilestone**, **UserBadge**, y **Configuration** también contienen claves foráneas que relacionan usuarios con sus respectivos ejercicios, hitos, insignias y configuraciones. En la sección superior se encuentran aquellas tablas que hemos llegado a implementar y en la parte inferior aquellas que no.

Exercise (id_exercise, exercise_name, difficulty, speed, experience);

RoutineExercise (id_routine*, id_exercise*, sequence_order);

Routine (id_routine, name, description, creation date, id_user*);

Users (id_user, username, email, level, experience, registration_date);

UserExercise (id_exercise*, id_user*, completion_date);

Milestones (id_milestone, name, description, reward, week)

UserMilestone(id_milestone*, id_user*, completition_date);

Badges (id_badge, name, description, criteria);

UserBadge (id_badge*, id_user*, completion_date);

Configuration (id_configuration , id_user*, difficulty, notifications, audio);

Explicación de Tablas

1. Exercise

Representa los ejercicios disponibles en el sistema que pueden ser realizados

- **Campos:**
  - *id_exercise* (PK): Identificador único de cada ejercicio.
  - *exercise_name*: Nombre del ejercicio.
  - *difficulty*: Nivel de dificultad del ejercicio (por ejemplo: fácil, medio, difícil).
  - *speed*: Velocidad requerida o sugerida para realizar el ejercicio.
  - *experience*: Puntos de experiencia otorgados al completar el ejercicio.
- **Relaciones:**
  - Relacionado con los ejercicios a través de *RoutineExercise* (1 ejercicio podrá estar en muchas rutinas).
  - Relacionado con los usuarios a través de *UserExercise* (1 ejercicio puede ser completado por varios usuarios).

2. RoutineExercise

Relaciona los ejercicios con las rutinas, indicando el orden de ejecución.

- **Campos:**
  - *id_routine** (PK, FK): Identificador de la rutina.
  - *id_exercise** (PK, FK): Identificador del ejercicio.
  - *sequence_order*: Orden de ejecución del ejercicio en la rutina.
- **Claves:**
  - Clave primaria compuesta: (*id_routine*, *id_exercise*).
- **Relaciones:**
  - Relacionado con *Routine* mediante *id_routine*.
  - Relacionado con *Exercise* mediante *id_exercise*.

3. Routine

Representa las rutinas diseñadas para los usuarios.

- **Campos:**
    - *id_routine* (PK): Identificador único de la rutina.
    - *name:* Nombre de la rutina.
    - *description:* Descripción de la rutina.
    - *creation_date*: Fecha en la que se creó la rutina.
    - *id_user** (FK): Identificador del usuario que creó la rutina.
- **Relaciones:**
    - Relacionado con *Users* mediante *id_user*.(La rutina solo permanecerá en principio a un usuario)
    - Relacionado con los ejercicios a través de *RoutineExercise.*

4. Users

Contiene los datos de los usuarios registrados en el sistema.

- **Campos:**
    - *id_user* (PK): Identificador único del usuario.
    - *username*: Nombre de usuario.
    - *email*: Correo electrónico del usuario.
    - *level*: Nivel actual del usuario.
    - *experience*: Puntos de experiencia acumulados por el usuario.
    - *hashed_password*: Contraseña hasheada del usuario.
    - token: El token que valida al usuario logueado.
    - *registration_date*: Fecha de registro del usuario.
- **Relaciones:**
    - Relacionado con *Routine* (1 usuario puede crear muchas rutinas).
    - Relacionado con ejercicios a través de *UserExercise* (1 usuario puede completar muchos ejercicios).
    - Relacionado con las metas a través de *UserMilestone* y *UserBadge* para registrar logros y recompensas.
    - Relacionado con *Configuration* (1 usuario tiene 1 configuración).

5. UserExercise

Registra qué ejercicios ha completado un usuario y cuándo.

- **Campos:**
  - *id_exercise** (PK, FK): Identificador del ejercicio completado.
  - *id_user** (PK, FK): Identificador del usuario.
  - *completion_date*: Fecha de finalización del ejercicio.
- **Relaciones:**
  - Relacionado con *Exercise* mediante *id_exercise*.
  - Relacionado con *Users* mediante *id_user*.

6. Milestones

Representa los hitos o logros que los usuarios pueden alcanzar semanalmente.

- **Campos:**
  - *id_milestone* (PK): Identificador único del hito.
  - *name*: Nombre del hito.
  - *description*: Descripción del hito.
  - *reward*: Recompensa asociada al hito.
  - *week*: Semana en la que se puede lograr el hito.
- **Relaciones:**
  - Relacionado con el usuario a través de *UserMilestone* (1 hito puede ser alcanzado por muchos usuarios).

7. UserMilestone

Relaciona a los usuarios con los hitos que han logrado.

- **Campos:**
  - *id_milestone** (PK, FK): Identificador del hito alcanzado.
  - *id_user** (PK, FK): Identificador del usuario.
  - *completion_date*: Fecha de finalización del hito.
- **Relaciones:**
  - Relacionado con *Milestones* mediante *id_milestone*.
  - Relacionado con *Users* mediante *id_user*.

8. Badges

Representa las insignias que los usuarios pueden ganar.

- **Campos:**
  - *id_badge* (PK): Identificador único de la insignia.
  - *name*: Nombre de la insignia.
  - *description*: Descripción de la insignia.
  - *criteria*: Criterio para obtener la insignia.
- **Relaciones:**
  - Relacionado con los usuarios a través de *UserBadge* (1 insignia puede ser otorgada a muchos usuarios).

9. UserBadge

Relaciona a los usuarios con las insignias que han ganado.

- **Campos:**
  - *id_badge** (PK, FK): Identificador de la insignia ganada.
  - *id_user** (PK, FK): Identificador del usuario.
  - *completion_date*: Fecha en la que se ganó la insignia.
- **Relaciones:**
  - Relacionado con *Badges* mediante *id_badge*.
  - Relacionado con *Users* mediante *id_user*.

10. Configuration

- **Descripción:** Almacena la configuración personalizada de cada usuario.
- **Campos:**
  - *id_configuration* (PK): Identificador único de la configuración.
  - *id_user** (FK): Identificador del usuario.
  - *difficulty*: Nivel de dificultad predeterminado en la configuración.
  - *notifications*: Preferencias de notificaciones.
  - *audio*: Configuración de audio.
- **Relaciones:**
  - Relacionado con *los usuarios* mediante *id_user*.

Claves Primarias y Foráneas

- Las claves primarias están indicadas como *(PK)*.
- Las claves foráneas están indicadas como *(FK)* y relacionan las tablas según las necesidades del modelo.

Relación General entre las Tablas

- *Users* es el núcleo principal que conecta con *Routine*, *UserExercise*, *UserMilestone*, *UserBadge*, y *Configuration*.
- *Exercise* se relaciona con *RoutineExercise* y *UserExercise*.
- *RoutineExercise* conecta *Routine* con *Exercise*.
- *Milestones* y *Badges* representan logros individuales relacionados con los usuarios mediante tablas intermedias.

Modelos -

Exercise -

El modelo **Exercise** representa una entidad de base de datos que contiene información sobre los ejercicios disponibles en el sistema. Este modelo está mapeado a la tabla `exercises` en la base de datos y tiene las siguientes características principales:

Anotaciones principales:

- *@Entity*: Indica que esta clase es una entidad JPA y está mapeada a una tabla de la base de datos.
- *@Table(name = "exercises")*: Especifica el nombre de la tabla asociada a esta entidad en la base de datos.

Campos y mapeos:

- *id*: Campo primario (*@Id*) generado automáticamente mediante la estrategia *IDENTITY* (gestión de la base de datos).
- *exercise_name*: Nombre del ejercicio, mapeado a la columna *exercise_name*.
- *difficulty*: Nivel de dificultad del ejercicio (entero), mapeado a la columna *difficulty*.
- *speed*: Velocidad asociada al ejercicio (tipo *Float*), mapeada a la columna *speed*.
- *experience*: Puntos de experiencia que otorga el ejercicio, mapeado a la columna *experience*.

Routine -

El modelo **Routine** representa la entidad que almacena información sobre las rutinas creadas en el sistema. Está mapeado a la tabla *routines* en la base de datos y contiene detalles como el nombre, descripción, fecha de creación, y el usuario que la creó.

Anotaciones principales:

- *@Entity*: Indica que esta clase representa una entidad JPA que se relaciona con una tabla de la base de datos.
- *@Table(name = "routines")*: Define que esta entidad está mapeada a la tabla *routines*.
- *@Id*: Marca el campo *id* como la clave primaria.
- *@GeneratedValue(strategy = GenerationType.IDENTITY)*: Especifica que el valor de `id` se genera automáticamente por la base de datos.
- *@CreationTimestamp*: Anotación que inicializa automáticamente el campo *creation_date* con la fecha y hora en que se crea la entidad.
- *@JsonProperty*: Permite personalizar cómo se serializa el JSON al interactuar con la API. En este caso, transforma los nombres de los campos al formato esperado por el cliente.

Campos y mapeos:

- **id**: Campo primario (*@Id*) generado automáticamente mediante la estrategia *IDENTITY* (gestión de la base de datos), mapeado a la columna *id_routine*.
- **routine_name**: Nombre de la rutina, mapeado a la columna *routine_name*. Serializado/deserializado en JSON como *routine_name*.
- **routine_description**: Descripción de la rutina, mapeada a la columna *routine_description*.
- **creation_date**: Fecha y hora de creación de la rutina, mapeada a la columna *creation_date*. Se inicializa automáticamente con la fecha y hora actuales mediante la anotación *@CreationTimestamp*.
- **id_user**: Identificador del usuario que creó la rutina, mapeado a la columna *id_user*.

User -

El modelo **User** representa la entidad que almacena información sobre los usuarios creados en el sistema. Está mapeado a la tabla *user* en la base de datos y contiene detalles como el username, email, fecha de creación, y su nivel.

Anotaciones principales:

- *@Entity*: Indica que esta clase representa una entidad JPA que se relaciona con una tabla de la base de datos.
- *@Table(name = "user")*: Define que esta entidad está mapeada a la tabla *routines*.
- *@Id*: Marca el campo *id* como la clave primaria.
- *@GeneratedValue(strategy = GenerationType.IDENTITY)*: Especifica que el valor de `id` se genera automáticamente por la base de datos.
- *@CreationTimestamp*: Anotación que inicializa automáticamente el campo *registration_date* con la fecha y hora en que se crea la entidad.
- *@JsonProperty*: Permite personalizar cómo se serializa el JSON al interactuar con la API. En este caso, transforma los nombres de los campos al formato esperado por el cliente.

Campos y mapeos:

- **id**: Campo primario (*@Id*) generado automáticamente mediante la estrategia *IDENTITY* (gestión de la base de datos), mapeado a la columna *id_user*.
- **username**: Nombre de usuario, mapeado a la columna *username*. Serializado/deserializado en JSON como *username*.
- **email**: Email que empleó el usuario para registrarse, mapeado a la columna *email*. Serializado/deserializado en JSON como *email*.
- **level**: Nivel del usuario en la aplicación, mapeado a la columna *level*. Serializado/deserializado en JSON como *level*.
- **experience**: Experiencia actual del usuario en la aplicación, mapeado a la columna *experience*. Serializado/deserializado en JSON como *experience*.
- **registration_date**: Fecha y hora de creación del usuario, mapeada a la columna *registration_date*. Se inicializa automáticamente con la fecha y hora actuales mediante la anotación *@CreationTimestamp*.
- **hashedPassword**: Contraseña hasheada del usuario, mapeado a la columna *hashedPassword*. Serializado/deserializado en JSON como *hashedPassword*.
- **token**: token generado para el usuario al loguearse, mapeado a la columna *token*. Serializado/deserializado en JSON como *token*.

Badge -

El modelo **Badge** representa la entidad que almacena información sobre las medallas creadas en el sistema. Está mapeado a la tabla *badge* en la base de datos y contiene detalles como el título, la descripción y su condición de desbloqueo.

Anotaciones principales:

- *@Entity*: Indica que esta clase representa una entidad JPA que se relaciona con una tabla de la base de datos.
- *@Table(name = "badge")*: Define que esta entidad está mapeada a la tabla *routines*.
- *@Id*: Marca el campo *id* como la clave primaria.
- *@GeneratedValue(strategy = GenerationType.IDENTITY)*: Especifica que el valor de `id` se genera automáticamente por la base de datos.
- *@JsonProperty*: Permite personalizar cómo se serializa el JSON al interactuar con la API. En este caso, transforma los nombres de los campos al formato esperado por el cliente.

Campos y mapeos:

- **id**: Campo primario (*@Id*) generado automáticamente mediante la estrategia *IDENTITY* (gestión de la base de datos), mapeado a la columna *id_badge*.
- **name**: Título de la medalla, mapeado a la columna *name*. Serializado/deserializado en JSON como *name*.
- **description**: descripción de dicha medalla, mapeado a la columna *description*. Serializado/deserializado en JSON como *description*.
- **criteria**: Criterio para obtener la medalla, mapeado a la columna *criteria*. Serializado/deserializado en JSON como *criteria*.

Milestone -

El modelo **Milestone** representa la entidad que almacena información sobre los hitos creados en el sistema. Está mapeado a la tabla *milestones* en la base de datos y contiene detalles como el título, objetivo, recompensa y su progreso actual.

Anotaciones principales:

- *@Entity*: Indica que esta clase representa una entidad JPA que se relaciona con una tabla de la base de datos.
- *@Table(name = "milestones")*: Define que esta entidad está mapeada a la tabla *milestones*.
- *@Id*: Marca el campo *id* como la clave primaria.
- *@GeneratedValue(strategy = GenerationType.IDENTITY)*: Especifica que el valor de `id` se genera automáticamente por la base de datos.
- *@JsonProperty*: Permite personalizar cómo se serializa el JSON al interactuar con la API. En este caso, transforma los nombres de los campos al formato esperado por el cliente.

Campos y mapeos:

- **id**: Campo primario (*@Id*) generado automáticamente mediante la estrategia *IDENTITY* (gestión de la base de datos), mapeado a la columna *id_milestone*.
- **title**: Título del objetivo, mapeado a la columna *title*. Serializado/deserializado en JSON como *title*.
- **obtention**: Objetivo a completar del hito, mapeado a la columna *obtention*. Serializado/deserializado en JSON como *obtention*.
- **reward**: Recompensa por completar el objetivo, mapeado a la columna *reward*. Serializado/deserializado en JSON como *reward*.
- **progress**: Progreso del objetivo, mapeado a la columna *progress*. Serializado/deserializado en JSON como *progress*.

UserExerciseKey -

El modelo **UserExerciseKey** es una clase embebible (*@Embeddable*) que define una clave compuesta para la tabla que relaciona *User* y *Exercise*. Esta clave permite que una combinación de *id_user* y *id_exercise* sea única en la tabla asociada.

Anotaciones principales:

- ***@Embeddable***: Marca esta clase como una clave embebible que será utilizada en otra entidad.
- **Implementación de *Serializable***: Permite que esta clase sea utilizada como clave primaria compuesta en entidades JPA.
- **Métodos *equals* y *hashCode***: Garantizan una correcta comparación y almacenamiento en estructuras como *HashMap* o *HashSet*.

Campos y mapeos:

- **idUser**: Identificador del usuario, mapeado a la columna *id_user*.
- **idExercise**: Identificador del ejercicio, mapeado a la columna *id_exercise*.

UserExercise -

El modelo **UserExercise** representa la relación muchos a muchos entre **User** y **Exercise**, utilizando una clave primaria compuesta. Además, incluye un campo adicional, *completition_date*, para especificar la fecha en la que se completó el ejercicio.

Anotaciones principales:

- **Relaciones**: Se conecta con las entidades **User** y **Exercise** mediante claves externas (*id_user* y *id_exercise*).
- **Clave compuesta**: Usa la clase *UserExerciseKey* para definir una clave primaria compuesta que garantiza la unicidad de la combinación de usuario y ejercicio.
- **Fecha de Completación**: Proporciona un campo adicional *completition_date* que permite definir la fecha en la que se completó el ejercicio.

Campos y mapeos:

- **id**: Clave primaria compuesta embebida, definida por la clase *UserExerciseKey*. Mapeada mediante la anotación *@EmbeddedId*.
- **user**: Referencia a la entidad **User**. Relación muchos a uno, mapeada por el campo *idUser* de la clave compuesta. Mapeado con *@ManyToOne* y *@MapsId* para enlazarla a la columna *id_user* en la tabla.
- **exercise**: Referencia a la entidad **Exercise**. Relación muchos a uno, mapeada por el campo *idExercise* de la clave compuesta. Mapeado con *@ManyToOne* y *@MapsId* para enlazarla a la columna *id_exercise* en la tabla.
- **completition_date**: Fecha de completación del ejercicio, mapeado a la columna *completition_date*.

RoutineExerciseKey -

El modelo **RoutineExerciseKey** es una clase embebible (*@Embeddable*) que define una clave compuesta para la tabla que relaciona *Routine* y *Exercise*. Esta clave permite que una combinación de *id_routine* y *id_exercise* sea única en la tabla asociada.

Anotaciones principales:

- ***@Embeddable***: Marca esta clase como una clave embebible que será utilizada en otra entidad.
- **Implementación de *Serializable***: Permite que esta clase sea utilizada como clave primaria compuesta en entidades JPA.
- **Métodos *equals* y *hashCode***: Garantizan una correcta comparación y almacenamiento en estructuras como *HashMap* o *HashSet*.

Campos y mapeos:

- **idRoutine**: Identificador de la rutina, mapeado a la columna *id_routine*.
- **idExercise**: Identificador del ejercicio, mapeado a la columna *id_exercise*.

RoutineExercise -

El modelo **RoutineExercise** representa la relación muchos a muchos entre **Routine** y **Exercise**, utilizando una clave primaria compuesta. Además, incluye un campo adicional, *sequence_order*, para especificar el orden en que los ejercicios aparecen dentro de una rutina.

Anotaciones principales:

- **Relaciones**: Se conecta con las entidades **Routine** y **Exercise** mediante claves externas (*id_routine* y *id_exercise*).
- **Clave compuesta**: Usa la clase *RoutineExerciseKey* para definir una clave primaria compuesta que garantiza la unicidad de la combinación de rutina y ejercicio.
- **Orden secuencial**: Proporciona un campo adicional *sequence_order* que permite definir un orden específico para los ejercicios en una rutina.

Campos y mapeos:

- **id**: Clave primaria compuesta embebida, definida por la clase *RoutineExerciseKey*. Mapeada mediante la anotación *@EmbeddedId*.
- **routine**: Referencia a la entidad **Routine**. Relación muchos a uno, mapeada por el campo *idRoutine* de la clave compuesta. Mapeado con *@ManyToOne* y *@MapsId* para enlazarla a la columna *id_routine* en la tabla.
- **exercise**: Referencia a la entidad **Exercise**. Relación muchos a uno, mapeada por el campo *idExercise* de la clave compuesta. Mapeado con *@ManyToOne* y *@MapsId* para enlazarla a la columna *id_exercise* en la tabla.
- **sequence_order**: Orden secuencial del ejercicio dentro de la rutina, mapeado a la columna *sequence_order*.

UserMilestoneKey -

El modelo **UserMilestoneKey** es una clase embebible (*@Embeddable*) que define una clave compuesta para la tabla que relaciona *User* y *Milestones*. Esta clave permite que una combinación de *id_user* y *id_milestone* sea única en la tabla asociada.

Anotaciones principales:

- ***@Embeddable***: Marca esta clase como una clave embebible que será utilizada en otra entidad.
- **Implementación de *Serializable***: Permite que esta clase sea utilizada como clave primaria compuesta en entidades JPA.
- **Métodos *equals* y *hashCode***: Garantizan una correcta comparación y almacenamiento en estructuras como *HashMap* o *HashSet*.

Campos y mapeos:

- **idUser**: Identificador del usuario, mapeado a la columna *id_user*.
- **idMilestone**: Identificador del milestone, mapeado a la columna *id_milestone*.

UserMilestone -

El modelo **UserMilestone** representa la relación muchos a muchos entre **User** y **Milestone**, utilizando una clave primaria compuesta. Además, incluye un campo adicional, *completion_status*, para especificar si ya se ha completado el Hito.

Anotaciones principales:

- **Relaciones**: Se conecta con las entidades **User** y **Milestone** mediante claves externas (*id_user* y *id_milestone*).
- **Clave compuesta**: Usa la clase *UserMilestoneKey* para definir una clave primaria compuesta que garantiza la unicidad de la combinación de usuario e hito.
- **Estado de Completación**: Proporciona un campo adicional *completion_status* que permite visualizar si el hito se completó o no.

Campos y mapeos:

- **id**: Clave primaria compuesta embebida, definida por la clase *RoutineMilestoneKey*. Mapeada mediante la anotación *@EmbeddedId*.
- **user**: Referencia a la entidad **User**. Relación muchos a uno, mapeada por el campo *idUser* de la clave compuesta. Mapeado con *@ManyToOne* y *@MapsId* para enlazarla a la columna *id_user* en la tabla.
- **milestone**: Referencia a la entidad **Exercise**. Relación muchos a uno, mapeada por el campo *idExercise* de la clave compuesta. Mapeado con *@ManyToOne* y *@MapsId* para enlazarla a la columna *id_exercise* en la tabla.
- **completition_status**: Estado de completación del hito, mapeado a la columna *completition_status*.

Repositorios -

El **JPA Repository** es una interfaz de Spring Data que simplifica las operaciones de acceso a datos en entidades JPA. Proporciona métodos predefinidos como *save*, *findById*, *delete*, y *findAll*, reduciendo el esfuerzo necesario para implementar las operaciones CRUD básicas. Además, permite la creación de métodos personalizados para consultas específicas, aprovechando la capacidad de Spring Data JPA para interpretar nombres de métodos y generar las consultas correspondientes.

En este caso, se utiliza el repositorio **RoutineExerciseRepository** para gestionar la relación entre **Routine** y **Exercise**. La clave primaria compuesta se define usando la clase *RoutineExerciseKey*, y los métodos personalizados facilitan consultas específicas para esta relación.

Métodos Personalizados -

Spring Data JPA interpreta *findByRoutine_Id* como una consulta que busca registros donde el campo `id` de la entidad *Routine* coincide con el valor proporcionado.

```
List<RoutineExercise> findByRoutine_Id(Long id);
```

Spring Data JPA interpreta *findByRoutine_IdAndExercise_Id* como una consulta compuesta que busca registros con una combinación de *id_routine* y *id_exercise*.

```
RoutineExercise findByRoutine_IdAndExercise_Id(Long id_routine, Long id_exercise);
```

Spring Data JPA interpreta *findByUserId* como una consulta que busca registros de la entidad Routine que coincide con el valor del *id_user* proporcionado.

```
List<Routine> findByUserId(@Param("userId") Integer userId);
```

Spring Data JPA interpreta *findByUsername* como una consulta que busca registros de la entidad User que coincide con el valor del *username* proporcionado.

```
Optional<User> findByUsername(String username);
```

Controladores -

Los controladores están diseñados para manejar solicitudes relacionadas con las entidades que tratan, nosotros tenemos 3 controladores, uno se encarga de los ejercicios, otro se encarga de las rutinas y el último maneja la relación entre ambos.

ExerciseController -

getExercises()

- Este método maneja solicitudes GET al endpoint base (/api/exercises).
- Recupera y devuelve una lista de todos los ejercicios almacenados en la base de datos.
- Retorna una lista de objetos **Exercise** (List<Exercise>).

getExerciseById(Long id)

- Este método maneja solicitudes GET a un endpoint específico con un identificador (/api/exercises/{id}). Recupera y devuelve un ejercicio en función de su id.
- **Retorna** un objeto **Exercise** si se encuentra el id, si el ejercicio no existe, se puede lanzar una excepción.

deleteExerciseByRoutineId(Long id_routine, Long id_exercise)

- Maneja solicitudes DELETE a un endpoint que incluye los identificadores de rutina y ejercicio (/api/exercises/{id_routine}/{id_exercise}).
- Busca una relación específica de **RoutineExercise** que conecta la rutina y el ejercicio proporcionados, si la relación existe, la elimina; de lo contrario, devuelve un error.

RoutineController -

getRoutines()

- Maneja solicitudes GET al endpoint base (/api/routines). Recupera y devuelve una lista de todas las rutinas almacenadas en la base de datos.
- Retorna una lista de objetos **Routine** (List<Routine>).

getRoutineById(Long id)

- Maneja solicitudes GET a un endpoint específico con un identificador (/api/routines/{id}). Recupera y devuelve una rutina en función de su id.
- Retorna un objeto **Routine** si se encuentra el id, si no existe, puede lanzarse una excepción.

getRoutinesByUserId(Long userid)

- Maneja solicitudes GET a un endpoint específico con un identificador (/api/routines/user/{userid}). Recupera y devuelve las rutinas en función del id del usuario.
- Retorna una lista de objetos **Routine** si se encuentran con el id del usuario, si no existe, puede lanzarse una excepción.

insertRoutine(Routine routine)

- Inserta una nueva rutina en la base de datos usando los datos proporcionados en el cuerpo de la solicitud.
- Devuelve un objeto **Routine** con la rutina creada.

updateRoutine(Long id, Routine newRoutine)

- Busca una rutina existente por su id. Si se encuentra, actualiza los campos routine_name, description e id_user con los datos de la nueva rutina proporcionada. Si no se encuentra, responde con **404 (Not Found)**.
- Devuelve el objeto **Routine** actualizado, estado HTTP **404 (Not Found)** si no existe la rutina.

deleteRoutine(Long id)

- Maneja solicitudes DELETE a un endpoint específico (/api/routines/{id}). Busca y elimina una rutina en la base de datos según su *id*.

UserController -

getUserByUsername(String username)

- Maneja solicitudes GET al endpoint base (/api/user/{username}).
- Recupera y devuelve al usuario deseado del registro de *User* almacenados en la base de datos.
- **Retorna** un objeto **User** si se encuentra uno con el {username}; si el usuario no existe, se puede lanzar una excepción.

updateUser(Long id, User userDetails)

- Maneja solicitudes PUT al endpoint base (/api/user/{id}).
- Busca y edita al usuario deseado del registro de *User* almacenados en la base de datos.
- **Retorna** un booleano verdadero si realiza el put; si no realiza el put,el booleano es falso.

addExperience(Long id, User userDetails)

- Maneja solicitudes PUT al endpoint base (/api/user/{id}/exp).
- Busca y edita al usuario, cambiando su experiencia y su nivel si la experiencia llega a un número determinado deseado, del registro de *User* almacenados en la base de datos.
- **Retorna** un booleano verdadero si realiza el put; si no realiza el put,el booleano es falso.

deleteUser(Long id)

- Maneja solicitudes Delete al endpoint base (/api/user/{id}).
- Busca y elimina al usuario deseado del registro de *User* almacenados en la base de datos.
- **Retorna** un booleano verdadero si realiza el delete; si no realiza el delete,el booleano es falso.

BadgeController -

getBadges()

- Maneja solicitudes GET al endpoint base (/api/badges). Recupera y devuelve una lista de todas las medallas almacenadas en la base de datos.
- **Retorna** una lista de objetos **Badge** (List<Badge>).

MilestoneController -

getMilestones()

- Maneja solicitudes GET al endpoint base (/api/milestones). Recupera y devuelve una lista de todos los hitos almacenados en la base de datos.
- **Retorna** una lista de objetos **Milestone** (List<Milestone>).

getMilestoneById(Long id)

- Maneja solicitudes GET al endpoint base (/api/milestones/{id}).
- Recupera y devuelve al hito deseado del registro de *Milestone* almacenados en la base de datos.
- **Retorna** un objeto **Milestone** si se encuentra uno con el {id}; si el hito no existe, se puede lanzar una excepción.

insertMilestone()

- Maneja solicitudes POST al endpoint base (/api/milestones).
- Inserta un nuevo hito en la base de datos usando los datos proporcionados en el cuerpo de la solicitud
- **Retorna** una solicitud con cuerpo, el cual contiene la información de si se ha generado el insert.

updateMilestone(Long id, Milestone newMilestone)

- Maneja solicitudes PUT al endpoint base (/api/milestones/{id}).
- Busca y edita el hito deseado del registro de *Milestone* almacenado en la base de datos.
- **Retorna** el objeto **Milestone** editado si realiza el put; si no realiza el put, devuelve una excepción.

deleteMilestone(Long id)

- Maneja solicitudes Delete al endpoint base (/api/milestones/{id}).
- Busca y elimina el hito deseado del registro de *Milestone* almacenado en la base de datos.
- **Retorna** un booleano verdadero si realiza el delete; si no realiza el delete,el booleano es falso.

RoutineExerciseController -

getAllRoutineExercises()

- Maneja solicitudes GET al endpoint base (/api/routine-exercise).
- Recupera y devuelve una lista de todos los registros de *RoutineExercise* almacenados en la base de datos.
- **Retorna** una lista de objetos **RoutineExercise** (List<RoutineExercise>).

findExercisesByRoutineId(Long id_routine)

- Recupera todos los ejercicios asociados a una rutina específica, ordenados por el campo *sequence_order*. Si no se encuentran ejercicios para la rutina especificada, devuelve una lista vacía.
- Devuelve una lista de mapas (List<Map<String, Object>>), donde cada mapa contiene los datos de un ejercicio asociado a la rutina, junto con su orden en la secuencia.

addExerciseToRoutine(Long routineId, Long exerciseId, Integer sequenceOrder)

- Permite agregar un ejercicio a una rutina específica, asignando un orden a este ejercicio dentro de la rutina.
- Crea un nuevo registro en la tabla *RoutineExercise* con las claves compuestas *routineId* y *exerciseId*. Si la operación es exitosa, guarda el nuevo registro y responde con el objeto creado.

UserExerciseController -

getAllUserExercises()

- Maneja solicitudes GET al endpoint base (/api/user-exercise).
- Recupera y devuelve una lista de todos los registros de *UserExercise* almacenados en la base de datos.
- **Retorna** una lista de objetos **UserExercise** (List<UserExercise>).

getUserExerciseById(Long userId, Long exerciseId)

- Recupera todos los ejercicios asociados a un usuario determinado. Si no se encuentran ejercicios para el usuario especificado, devuelve una lista vacía.
- Devuelve una lista de mapas (List<Map<String, Object>>), donde cada mapa contiene los datos de un ejercicio asociado al usuario.

createUserExercise(UserExercise userExercise)

- Permite crear una relación de un ejercicio a un usuario específico, manejando las solicitudes POST al endpoint base (/api/user-exercise).
- Crea un nuevo registro en la tabla *UserExercise* con las claves compuestas *userId* y *exerciseId*. Si la operación es exitosa, guarda el nuevo registro y responde con el objeto creado.

updateUserExercise(Long userId, Long exerciseId, UserExercise userExerciseDetails)

- Maneja las solicitudes PUT al endpoint base (/api/user-exercise/{userId}/{exerciseId}).
- Actualiza el registro en la tabla *UserExercise* con las claves compuestas *userId* y *exerciseId*. Si la operación es exitosa, guarda el registro editado y responde con el objeto editado.

deleteUserExercise(Long userId, Long exerciseId)

- Manejando las solicitudes PUT al endpoint base (/api/user-exercise/{userId}/{exerciseId}).
- Borra un registro de la tabla *UserExercise* con las claves compuestas *userId* y *exerciseId*. Si la operación es exitosa, devuelve un booleano verdadero y si no devuelve uno falso.

UserMilestoneController -

getAllUserMilestones()

- Maneja solicitudes GET al endpoint base (/api/user-milestones).
- Recupera y devuelve una lista de todos los registros de *UserMilestone* almacenados en la base de datos.
- **Retorna** una lista de objetos **UserMilestone** (List<UserMilestone>).

getUserMilestoneById(Long idUser, Long idMilestone)

- Recupera todos los ejercicios asociados a un usuario determinado. Si no se encuentran hitos para el usuario especificado, devuelve una lista vacía.
- Devuelve una lista de mapas (List<Map<String, Object>>), donde cada mapa contiene los datos de un hito asociado al usuario.

createUserMilestone(UserMilestone userMilestone)

- Permite crear una relación de un hito a un usuario específico, manejando las solicitudes POST al endpoint base (/api/user-milestones).
- Crea un nuevo registro en la tabla *UserMilestone* con las claves compuestas *idUser* y *idMilestone*. Si la operación es exitosa, guarda el nuevo registro y responde con el objeto creado.

updateUserMilestone(Long idUser, Long idMilestone, UserMilestone updatedUserMilestone)

- Maneja las solicitudes PUT al endpoint base (/api/user-milestone/{idUser}/{idMilestone}).
- Actualiza el registro en la tabla *UserMilestone* con las claves compuestas *idUser* y *idMilestone*. Si la operación es exitosa, guarda el registro editado y responde con el objeto editado.

deleteUserMilestone(Long idUser, Long idMilestone)

- Manejando las solicitudes PUT al endpoint base (/api/user-milestone/{idUser}/{idMilestone}).
- Borra un registro de la tabla *UserMilestone* con las claves compuestas *idUser* y *idMilestone*. Si la operación es exitosa, devuelve un booleano verdadero y si no devuelve uno falso.

Servicios -

La implementación hemos tenido que realizarla tanto para React como para Ionic/Angular, pero nos vamos a centrar en la implementación realizada en React.

ExerciseService -

getExercises

Este método realiza una solicitud ***GET*** a la API para obtener una lista de todos los ejercicios. El tipo de respuesta esperado es un arreglo de objetos ***Exercise***. Si la solicitud es exitosa, devuelve los datos (ejercicios) obtenidos; si ocurre un error, lo lanza para que sea manejado adecuadamente en el componente o servicio que lo invoque.

getExerciseById

Este método realiza una solicitud ***GET*** para obtener un ejercicio específico a través de su ID. Se pasa un parámetro ***id*** como parte de la URL. Si la solicitud tiene éxito, devuelve el ejercicio correspondiente al ***id*** solicitado; de lo contrario, lanza un error.

deleteExerciseById

Este método realiza una solicitud ***DELETE*** a la API para eliminar un ejercicio de una rutina, dado un ***id_routine*** y un ***id_exercise***. La URL de la solicitud incluye ambos identificadores. Si la eliminación es exitosa, devuelve los datos del ejercicio eliminado; si se produce un error, lo lanza para que se pueda manejar de manera adecuada.

RoutineService -

getRoutines

Realiza una solicitud ***GET*** a la API para obtener todas las rutinas. La respuesta esperada es una lista de objetos ***Routine***. Si la solicitud es exitosa, devuelve la lista de rutinas; si ocurre un error, lo lanza para su manejo en el componente o servicio que lo invoque.

getRoutineById

Realiza una solicitud ***GET*** a la API para obtener una rutina específica por su ***id***. Se pasa el parámetro ***id*** en la URL. Si la solicitud tiene éxito, devuelve la rutina correspondiente al ***id***; si ocurre un error, se lanza una excepción.

getRoutinesByUserId

Realiza una solicitud ***GET*** a la API para obtener unas rutinas específicas por la id del usuario. Se pasa el parámetro ***id*** del usuario en la URL. Si la solicitud tiene éxito, devuelve las rutinas correspondiente a la ***id del usuario***; si ocurre un error, se lanza una excepción.

addRoutine

Realiza una solicitud ***POST*** a la API para agregar una nueva rutina. Recibe el objeto ***Routine*** en el cuerpo de la solicitud. Si la creación de la rutina es exitosa, devuelve el objeto de la rutina creada; en caso de error, lo lanza.

updateRoutine

Realiza una solicitud ***PUT*** a la API para actualizar una rutina existente. Recibe el ***id*** de la rutina y el objeto ***Routine*** con los nuevos datos. Si la actualización es exitosa, devuelve la rutina actualizada; si se produce un error, se lanza.

deleteRoutine

Realiza una solicitud ***DELETE*** a la API para eliminar una rutina específica mediante su ***id***. Si la eliminación es exitosa, no retorna datos, pero si ocurre un error, lo lanza para que sea manejado adecuadamente.

UserService -

register

Realiza una solicitud ***POST*** a la API para crear un nuevo usuario con los datos enviados en el cuerpo de la solicitud. La respuesta esperada es un mensaje con dos campos para especificar si la solicitud es exitosa o no.

login

Realiza una solicitud ***POST*** a la API enviando en su cuerpo el usuario y su contraseña para realizar el login. Si la solicitud tiene éxito, devuelve el usuario y se guarda en el sessionstorage para mantener el login; si ocurre un error, se lanza una excepción.

checkBackendConnection

Realiza una solicitud ***GET*** a la API para comprobar el estado del backend. La función devolverá un booleano verdadero si recibe una respuesta del backend, en caso de que el backend no responda el booleano será falso.

getUserByUsername

Realiza una solicitud ***GET*** a la API para obtener un usuario mediante su ***username***. Recibe el ***username*** del usuario. Si se encuentra un usuario con ese ***username*** la api devolverá el objeto de **User**. Si ocurre un error, se lanza una excepción.

updateUserExp

Realiza una solicitud ***PUT*** a la API para modificar la experiencia del usuario. Si la modificación es exitosa, se devuelve un booleano verdadero, pero si ocurre un error, lo lanza para que sea manejado adecuadamente.

BadgeService -

getBadges

Realiza una solicitud ***GET*** a la API para recibir todas las medallas del usuario mediante su ***username***. La respuesta esperada es una lista de objetos ***badges***. En caso de error, retorna el error para manejarlo.

RoutineExerciseService -

getExerciseByRoutineId

Realiza una solicitud ***GET*** a la API para obtener todos los ejercicios asociados a una rutina específica, usando el ***id*** de la rutina. La respuesta esperada es una lista de objetos ***ExerciseResponse***. Si la solicitud es exitosa, devuelve los ejercicios asociados a la rutina; si ocurre un error, retorna ***null***.

addExerciseToRoutine

Realiza una solicitud ***POST*** a la API para agregar un ejercicio a una rutina específica. Requiere tres parámetros: ***routineId*** (ID de la rutina), ***exerciseId*** (ID del ejercicio), y ***sequenceOrder*** (el orden de secuencia en que se debe ejecutar el ejercicio). Si la operación es exitosa, devuelve la respuesta correspondiente; si ocurre un error, lo registra en la consola y lanza el error para su posterior manejo.

Implementación -

Exercise Selector -

fetchExercises

Esta función llama al método ***getExercises*** del servicio ***ExerciseService***. El propósito de esta función es obtener todos los ejercicios disponibles desde la API y almacenarlos en el estado ***exercises*** utilizando el hook ***useState***. Si ocurre un error durante la solicitud, se muestra un mensaje de error en la consola.

fetchRoutines

Similar a la función ***fetchExercises***, esta función hace uso del método ***getRoutines*** de ***RoutineService***. Recupera todas las rutinas almacenadas en la base de datos y actualiza el estado ***routines***. En caso de error, se maneja el problema imprimiendo un mensaje de error en la consola.

handleDeleteRoutine

Este método utiliza el servicio ***RoutineService*** para eliminar una rutina específica de la base de datos. Recibe el ***id*** de la rutina a eliminar, llama al método ***deleteRoutine***, y luego vuelve a cargar las rutinas llamando a ***fetchRoutines***. Si ocurre un error, este se captura y se muestra en la consola.

Routines -

fetchExercises

Esta función se encarga de recuperar los ejercicios asociados a una rutina. Si la rutina no es nueva (es decir, tiene un ID), llama al servicio **RoutineExerciseService.getExerciseByRoutineId**, pasando el ID de la rutina. Luego, se establece el estado de los ejercicios cargados en los estados ***exercises*** y ***tempExercises***.

handleAddExercise

Este método permite agregar ejercicios a una rutina. Llama al servicio **ExerciseService.getExercises** para obtener todos los ejercicios disponibles, luego filtra aquellos que no estén ya presentes en la lista ***tempExercises*** (ejercicios temporales que el usuario está agregando a la rutina). Si hay ejercicios disponibles para agregar, agrega el primero de la lista a ***tempExercises***.

handleRemoveTempExercise

Esta función permite eliminar un ejercicio de la lista temporal de ejercicios (***tempExercises***). Recibe el ***id_exercise*** del ejercicio a eliminar y actualiza el estado para reflejar la eliminación.

handleSubmit

Este método se ejecuta cuando el usuario guarda la rutina. Si la rutina es nueva (***isNew*** es ***true***), se crea una nueva rutina a través de **RoutineService.addRoutine** y luego se agregan los ejercicios a esa rutina utilizando los métodos ***addAllNewExercises***. Si la rutina ya existe, se actualiza con **RoutineService.updateRoutine**. Además, se gestionan los ejercicios de la rutina existente para agregar nuevos y eliminar los que ya no están en la lista.

addAllNewExercises

Esta función toma los ejercicios temporales (***tempExercises***) y los agrega a la rutina a través del servicio **RoutineExerciseService.addExerciseToRoutine**, asignando un orden de secuencia para cada ejercicio.

addNewExercises

Similar a ***addAllNewExercises***, pero en este caso solo agrega los ejercicios que no están ya presentes en la rutina. Primero compara los ejercicios existentes con los nuevos para asegurarse de no duplicar los ejercicios.

handleDeleteExercise

Cuando se desea eliminar un ejercicio de la rutina, este método llama al servicio **ExerciseService.deleteExerciseById**, pasando el ***id_exercise*** y el ***id_routine***, lo que elimina el ejercicio de la rutina en la base de datos.

getExistingRoutineExercises

Esta función obtiene los ejercicios ya asociados a una rutina existente usando **RoutineExerciseService.getExerciseByRoutineId**. Este es un paso previo a la actualización de la rutina, donde se determinan qué ejercicios eliminar o agregar.

Webgrafía -

React Documentation - Documentación oficial de React, contiene guías, tutoriales y la referencia completa de la API de React.

Axios Documentation - Documentación oficial de Axios, una biblioteca para realizar solicitudes HTTP.

React Router Documentation - Guía oficial de React Router, una librería para manejar rutas y navegación en aplicaciones React.

Ant Design Documentation - Documentación de Ant Design, una biblioteca de componentes UI para React, que incluye el componente Select usado en tu proyecto.

FontAwesome Icons - Repositorio de iconos gratuitos y premium, utilizados en el proyecto.

JavaScript Fetch API Documentation - Guía y documentación oficial de la API Fetch de JavaScript para realizar solicitudes HTTP.