Clase grabada

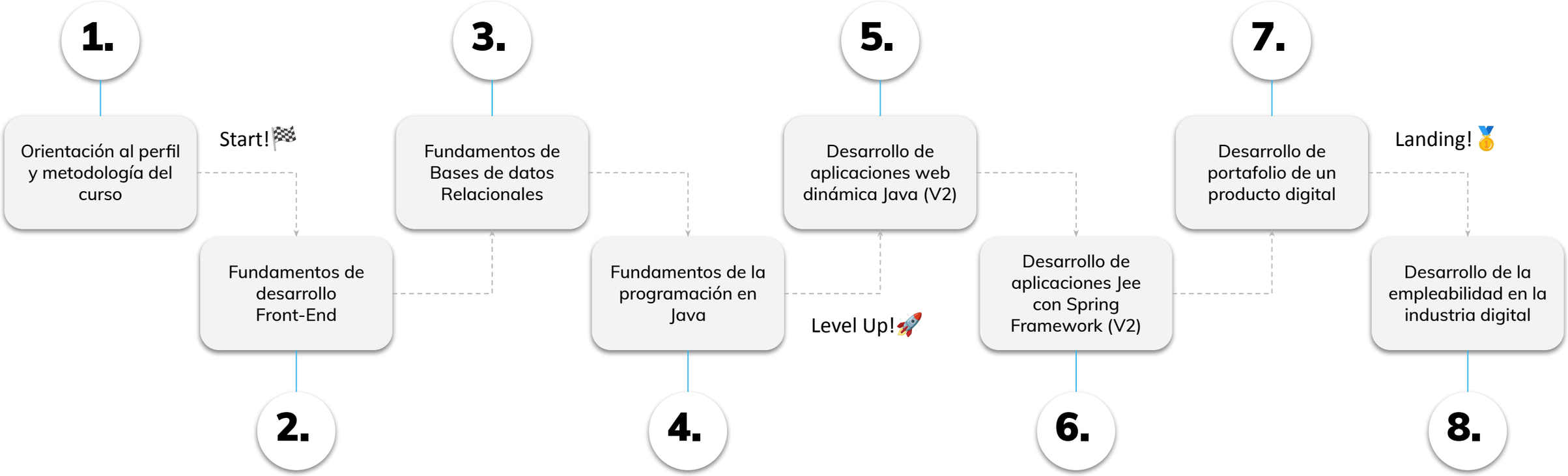
Acceso a datos en Spring Framework

Plan formativo: Desarrollo de Aplicaciones Full Stack Java Trainee V2.0

English Always | alkemy

HOJA DE RUTA

¿Cuáles **skill** conforman el programa?



English Always | alkemy

REPASO CLASE ANTERIOR

En la clase anterior trabajamos 📚:

- ✓ *Clase de repositorio con JPA*
- ✓ *Métodos heredados de CrudRepository*

LEARNING PATHWAY

¿Sobre qué temas trabajaremos?

6.3 Start! 🏁

Acceso a datos en Spring Framework

Relaciones de Asociación

Uno a Muchos

Asociaciones (uno a uno, uno a muchos)
Invocar un repositorio desde un servicio

English Always | alkemy

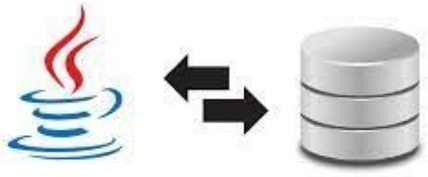
OBJETIVOS DE APRENDIZAJE

¿Qué aprenderemos?

- ***Implementar relaciones entre clase mediante JPA***

Asociaciones

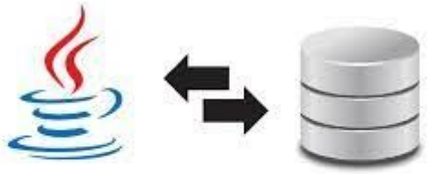
Asociaciones



JAVA PERSISTENCE API

Las asociaciones entre entidades son un aspecto clave en el modelado de bases de datos relacionales mediante Java Persistence API (JPA). Estas asociaciones definen cómo se relacionan las entidades entre sí, permitiendo representar conexiones lógicas y estructuras más complejas en una base de datos

Asociaciones



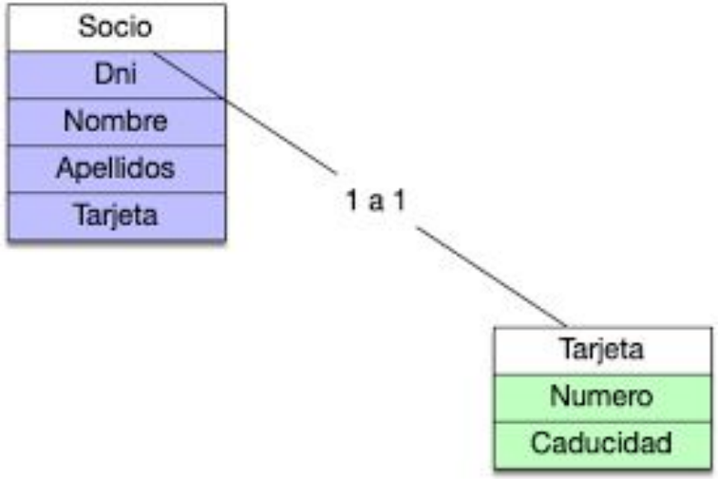
JAVA PERSISTENCE API

JPA nos brinda anotaciones para definir una relación entre dos clases que deseamos reflejar en la base de datos. Éstas anotaciones sirven para informarle a la base de datos cuál será el tipo de relación que tendrán las tablas entre sí. Recordemos que las anotaciones cumplen el propósito de “traducir” nuestro código de Java para que lo entienda la base de datos.

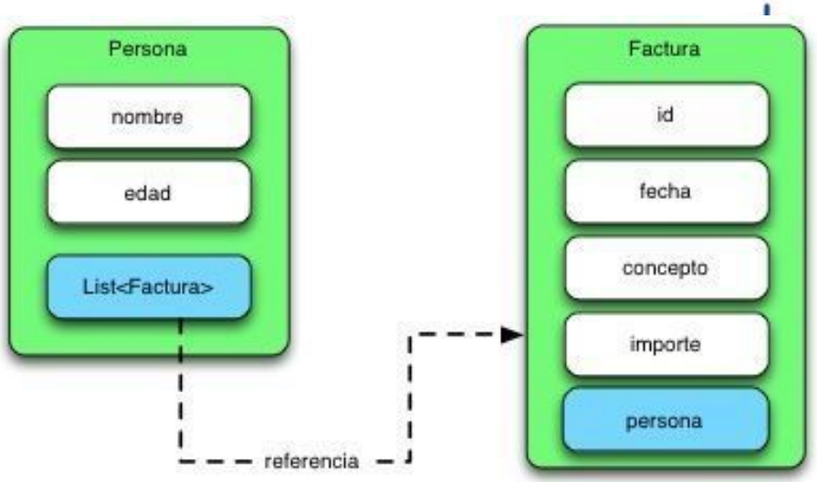
Asociaciones

Las anotaciones de asociación que nos provee JPA son:

@OneToOne: relación entre tablas uno a uno



@OneToMany: relación entre tablas uno a muchos.



Asociación Uno a Uno

La relación **One-to-One** (uno a uno) en JPA establece una asociación entre dos entidades donde una instancia de una entidad se relaciona con exactamente una instancia de otra entidad. En la entidad principal, se utiliza la anotación @OneToOne junto con @JoinColumn para especificar la columna que almacena la clave externa.

```java
@Entity
public class Usuario {

    @Id
    @GeneratedValue(generator = "uuid")
    private String id;
    private String nombre;
    private String email;

    @OneToOne
    @JoinColumn(name = "cuenta_id")
    private Cuenta cuenta;
```

Asociación Uno a Uno

En la entidad asociada, se utiliza @OneToOne(mappedBy = "nombrePropiedad") para establecer la relación **bidireccional**. Esta relación permite acceder a la entidad asociada desde la entidad principal y viceversa.

```java
@Entity
public class Cuenta {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @OneToOne(mappedBy = "usuario")
    private Usuario usuario;
```

Asociación Uno a Muchos

La relación **One-to-Many** en JPA establece una asociación **entre dos entidades donde una entidad principal tiene una colección de entidades asociadas**. En la entidad principal, se utiliza la anotación @OneToMany para definir la relación, y en la entidad asociada no es necesario determinar ninguna anotación para establecer la relación con la entidad principal.

Asociación Uno a Muchos

De esta manera, se crea una relación **One-to-Many** donde un Usuario puede tener múltiples Cuentas.

Por otro lado, la entidad Cuenta **no tiene una referencia directa** a la entidad Usuario. La relación se establece a través de la columna cuenta_id en la tabla Usuario.

```java
@Entity
public class Usuario {

    @Id
    @GeneratedValue(generator = "uuid")
    private String id;
    private String nombre;
    private String email;

    @OneToMany
    private List<Cuenta> listaCuentas;

```

Asociaciones

Consideraciones importantes:

Bidireccional vs. Unidireccional: Las asociaciones pueden ser bidireccionales (ambas entidades tienen referencias entre sí) o unidireccionales (una entidad tiene una referencia a la otra). La elección depende de los requisitos específicos de la aplicación.

Optimización de Consultas: Al trabajar con asociaciones, es importante considerar la optimización de consultas, ya que las consultas relacionadas pueden afectar significativamente el rendimiento. El uso adecuado de FetchType y la comprensión de la carga diferida pueden ayudar a evitar problemas de rendimiento.

Asociaciones

Consideraciones importantes:

Fetch Type: El atributo fetch en las anotaciones @OneToOne y @OneToMany define cómo se deben cargar los datos asociados. El valor predeterminado es FetchType.LAZY, lo que significa que la carga se realiza de forma diferida. Si se utiliza FetchType.EAGER, los datos asociados se cargarán de inmediato.

```java
@Entity
public class Usuario {

    @Id
    @GeneratedValue(generator = "uuid")
    private String id;
    private String nombre;
    private String email;

    @OneToOne(fetch = FetchType.LAZY)
    private Cuenta cuenta;
```

Evaluación Integradora ✨

¿Listos para un nuevo desafío? En esta clase comenzamos a construir nuestro....

Trabajo Integrador del Módulo 💪🏽

Iremos completándolo progresivamente clase a clase.

English Always | alkemy

LIVE CODING

Ejemplo en vivo

Completando la Wallet:

*Es momento de vincular nuestras entidades para **relacionar a cada usuario con su cuenta bancaria**.*

Para esto vamos a realizar algunos cambios:

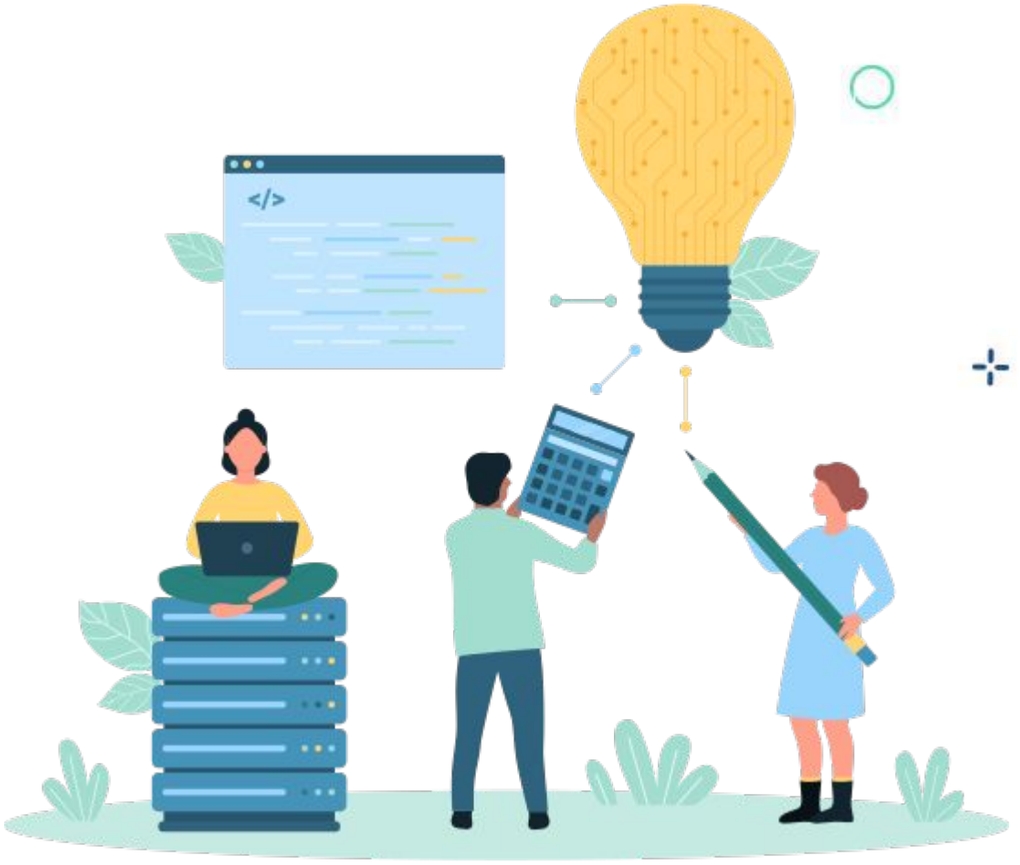
- ***Eliminar** las clases "DAO" creadas con la implementación de JDBCTemplate (**ya que utilizaremos los repositorios JPA**).*
- *En la entidad Usuario, agregar el atributo Cuenta, con una relación uno-a-uno y un atributo de carga EAGER.*

Tiempo: 15 minutos

Invocar un repositorio desde un servicio

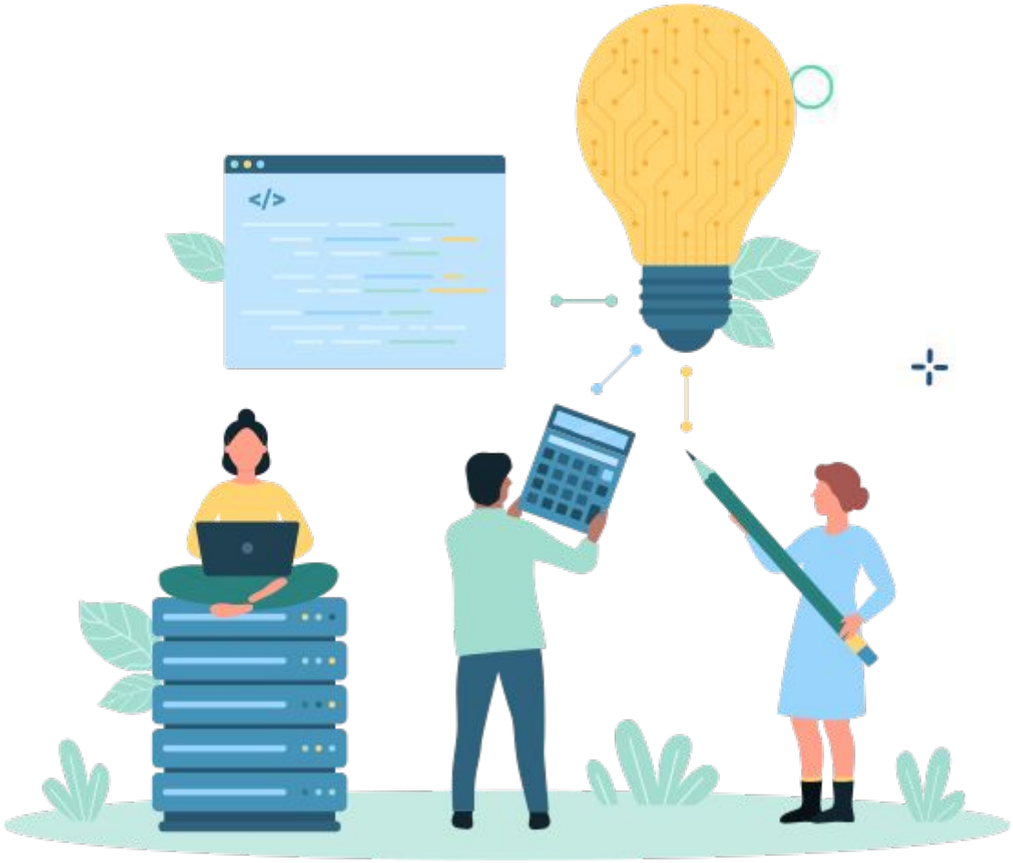
Invocar un repositorio desde un servicio

Invocar un repositorio desde un servicio es una práctica común en el desarrollo de aplicaciones Spring con JPA. Los servicios actúan como una capa intermedia entre el controlador (o cualquier otra capa de presentación) y el repositorio, encapsulando la lógica de negocio y permitiendo la interacción con la capa de acceso a datos.



Invocar un repositorio desde un servicio

Básicamente al invocar el repositorio desde el servicio puedes acceder a todas las operaciones definidas en el repositorio, como consultas personalizadas, operaciones de persistencia, eliminación, etc. Esto te permite encapsular la lógica de acceso a datos en el repositorio y utilizarla de manera conveniente desde el servicio.



Invocar un repositorio desde un servicio

¿Cómo invocar un repositorio desde el servicio?

1- **Inyección de Dependencia**: La anotación @Autowired se utiliza para inyectar la implementación de repositorio en el servicio. Esta es una práctica común en Spring para lograr la inversión de control y la inyección de dependencias. Veamos un ejemplo:

```
@Service
public class UsuarioServicio {

    @Autowired
    private final UsuarioRepositorio usuarioRepositorio;


    public UsuarioServicio(UsuarioRepositorio usuarioRepositorio) {
        this.usuarioRepositorio = usuarioRepositorio;
    }
```

Invocar un repositorio desde un servicio

2- **Operaciones con el repositorio:** Es hora de crear los métodos encargados de invocar al repositorio para que realice los cambios persistentes en la base de datos. Por ejemplo:

obtenerUsuarioPorId(Long id): Utiliza el método findById del repositorio para recuperar un usuario por su identificador.

obtenerUsuarioPorNombre(**String nombre)**: Utiliza un método personalizado del repositorio (buscarPorNombre) para recuperar un usuario por su nombre.

```java
@Service
public class UsuarioServicio {

    @Autowired
    private UsuarioRepositorio usuarioRepositorio;

    public Usuario obtenerUsuarioPorId(Long id) {
        return usuarioRepositorio.findById(id).orElse(null);
    }

    public Usuario obtenerUsuariosPorNombre(String nombre) {
        return usuarioRepositorio.buscarPorNombre(nombre);
    }

    public void guardarUsuario(Usuario usuario) {
        usuarioRepositorio.save(usuario);
    }

    public void eliminarUsuario(Long id) {
        usuarioRepositorio.deleteById(id);
    }

    public List<Usuario> obtenerUsuarios() {
        return usuarioRepositorio.findAll();
    }
}
```

Invocar un repositorio desde un servicio

guardarUsuario(Usuario usuario): Utiliza el método save del repositorio para guardar un usuario en la base de datos. La implementación de save determinará si se trata de una operación de creación o actualización.

eliminarUsuario(Long id): Utiliza el método deleteById del repositorio para eliminar un usuario por su identificador.

obtenerUsuarios(): Utiliza el método findAll del repositorio para recuperar una lista de usuarios.

```java
@Service
public class UsuarioServicio {

    @Autowired
    private UsuarioRepositorio usuarioRepositorio;

    public Usuario obtenerUsuarioPorId(Long id) {
        return usuarioRepositorio.findById(id).orElse(null);
    }

    public Usuario obtenerUsuariosPorNombre(String nombre) {
        return usuarioRepositorio.buscarPorNombre(nombre);
    }

    public void guardarUsuario(Usuario usuario) {
        usuarioRepositorio.save(usuario);
    }

    public void eliminarUsuario(Long id) {
        usuarioRepositorio.deleteById(id);
    }

    public List<Usuario> obtenerUsuarios() {
        return usuarioRepositorio.findAll();
    }
}
```

Invocar un repositorio desde un servicio

Lógica de Negocio Adicional:

El servicio puede contener lógica de negocio adicional según los requisitos de la aplicación. Por ejemplo, se podrían agregar métodos que realizan operaciones más complejas que involucran la manipulación de varios objetos Usuario o la aplicación de reglas de negocio específicas.

```java
@Service
public class UsuarioServicio {

    @Autowired
    private UsuarioRepositorio usuarioRepositorio;

    public Usuario obtenerUsuarioPorId(Long id) {
        return usuarioRepositorio.findById(id).orElse(null);
    }

    public Usuario obtenerUsuariosPorNombre(String nombre) {
        return usuarioRepositorio.buscarPorNombre(nombre);
    }

    public void guardarUsuario(Usuario usuario) {
        usuarioRepositorio.save(usuario);
    }

    public void eliminarUsuario(Long id) {
        usuarioRepositorio.deleteById(id);
    }

    public List<Usuario> obtenerUsuarios() {
        return usuarioRepositorio.findAll();
    }
}
```

Invocar un repositorio desde un servicio

Lógica de Negocio Adicional:

Por ejemplo este método **guardarUsuario** que recibe como parámetro el nombre y el email.

El método se encarga de realizar la lógica correspondiente y luego se llama al repositorio para que persista los cambios en la base de datos.

```
@Autowired
private UsuarioRepositorio usuarioRepositorio;

public void guardarUsuario(String nombre, String email) {
    Usuario usuario = new Usuario();

    usuario.setNombre(nombre);
    usuario.setEmail(email);

    usuarioRepositorio.save(usuario);
}
```

LIVE CODING

Ejemplo en vivo

Completando la Wallet: Es momento de operar en la base de datos con JPA.

- *Vamos a crear los métodos CRUD en los servicios de Usuario y de Cuenta y a invocar al repositorio de cada entidad para poder persistir los cambios en una base de datos.*

Tiempo: 30 minutos

English Always | alkemy

Momento:

Time-out!

⏰5 min.

Ejercicio N° 1

Uno a Muchos

Uno a Muchos

Contexto: 🙌

Vamos a continuar con el ejercicio de la clase anterior. En este caso vamos agregar una relación entre clases y a mapear con la anotación @OneToMany de JPA.

Consigna: ✍️

Agregar un atributo de clase que sea una lista y anotar la relación como @OneToMany

Tiempo🕐: *15 minutos*

¿Alguna consulta?

RESUMEN

¿Qué logramos en esta clase?

- ✓ ***Comprender las relaciones de asociación entre clases.***
- ✓ ***Aprender a invocar al repositorio desde un servicio.***

English Always | alkemy

#WorkingTime

Continuemos ejercitando

¡Antes de cerrar la clase! Te invitamos a: 👇👇👇

1. Repasar nuevamente la grabación de esta clase
2. Revisar el material compartido en la plataforma de Moodle (lo que se vio en clase y algún ejercicio adicional)
   a. *Lectura Modulo 6, Lección 3: páginas 19 - 22*
3. Traer al próximo encuentro, todas tus dudas y consultas para verlas antes de iniciar nuevo tema.

¡Muchas Gracias!

Nos vemos en la próxima clase 👋🏻

English Always | alkemy