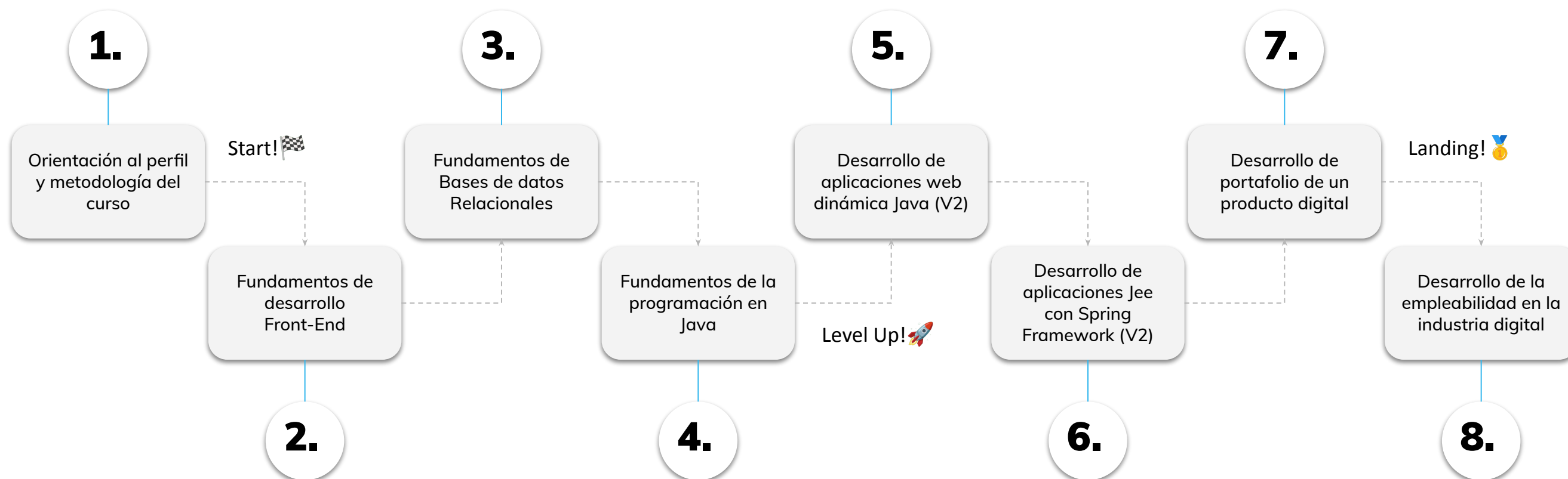


➤ Acceso a datos en Spring Framework

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

HOJA DE RUTA

¿Cuáles **skill** conforman el programa?



REPASO CLASE ANTERIOR

En la clase anterior trabajamos :

- ✓ Acceso y manipulación de datos desde el DAO
- ✓ Invocación del DAO desde un servicio

LEARNING PATHWAY

¿Sobre qué temas trabajaremos?

6.3

Start! 🏁

Acceso a datos en Spring Framework

Acceso a datos mediante JPA
La API de Persistencia de Java (JPA)
Clases de Entidad en JPA
El Entity Manager en JPA

JPA

Entidades

OBJETIVOS DE APRENDIZAJE

¿Qué aprenderemos?

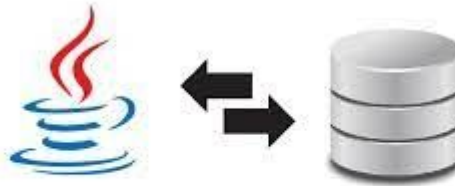


***Manipular la capa de acceso a datos
mediante JPA***



➤ Acceso a datos con JPA

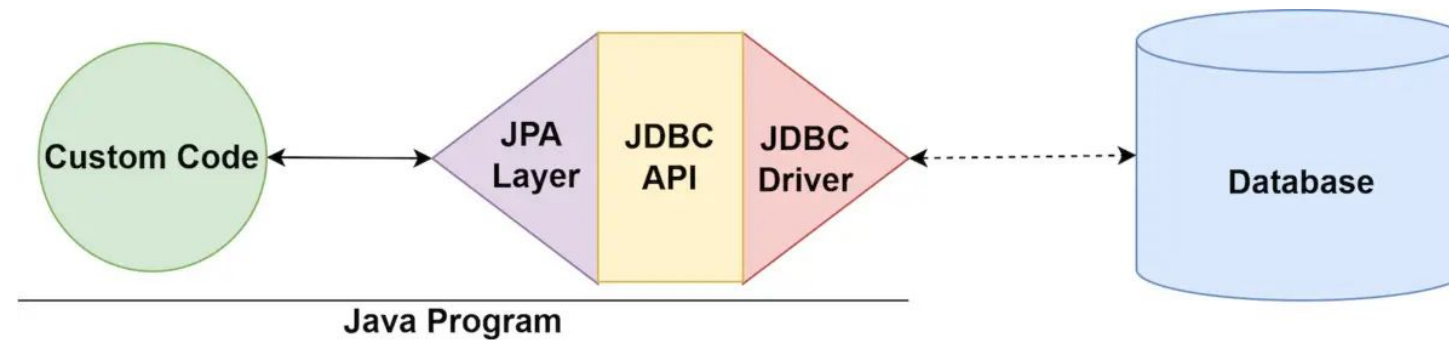
Acceso a datos con JPA



JAVA PERSISTENCE API

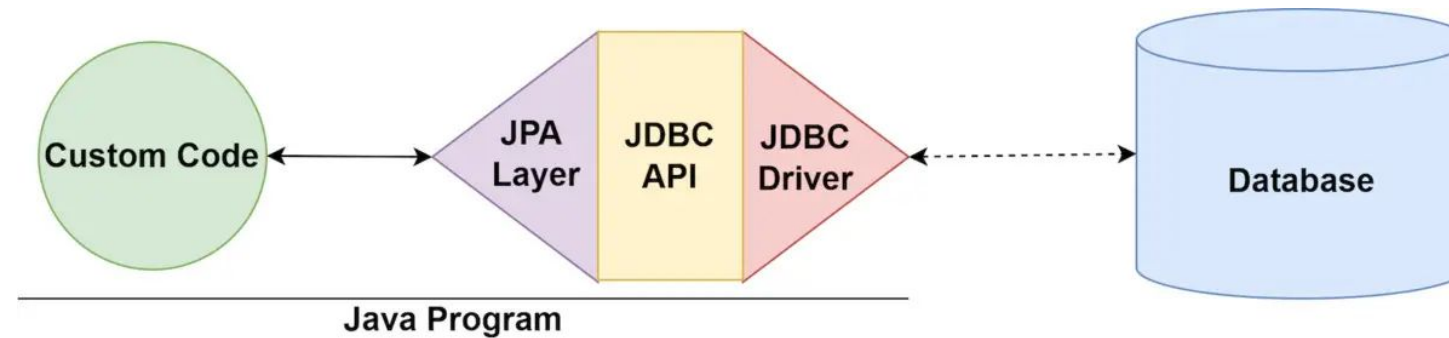
JPA (**J**ava **P**ersistence **A**PI) es la propuesta estándar que ofrece Java para implementar un Framework Object Relational Mapping (**ORM**), que **permite interactuar con la base de datos por medio de objetos**. JPA es el encargado de convertir los objetos Java en instrucciones para el Manejador de Base de Datos (DBMS).

Acceso a datos con JPA



JPA crea un estándar para la importante tarea de la correlación relacional de objetos mediante la utilización de **anotaciones** o **XML** para relacionar objetos con una o más tablas de una base de datos.

Acceso a datos con JPA



El objetivo que persigue el diseño de esta API es no perder las ventajas de la orientación a objetos al interactuar con una base de datos (siguiendo el patrón de mapeo objeto-relacional), como sí pasaba con EJB2, y permitir usar objetos regulares (conocidos como POJO).



Acceso a datos con JPA

Objetivos de JPA:



- **Abstracción de la Capa de Datos:** Esto permite a los desarrolladores trabajar con objetos Java en lugar de consultas SQL, facilitando el desarrollo y mantenimiento del código.
- **Portabilidad:** Los desarrolladores pueden escribir código independiente de la base de datos, y JPA se encargará de traducir las operaciones a la sintaxis específica de cada proveedor.
- **Mapeo Objeto-Relacional (ORM):** JPA facilita la tarea de mapear objetos Java a tablas de bases de datos y viceversa. Esto reduce la complejidad de las consultas SQL y mejora la legibilidad del código.
- **Transacciones:** JPA gestiona automáticamente las transacciones, asegurando la consistencia de los datos y proporcionando un mecanismo robusto para el control de concurrencia.





Acceso a datos con JPA

Características principales:




Anotaciones: JPA utiliza anotaciones Java para mapear entidades a tablas y definir relaciones entre ellas. Esto simplifica el proceso de configuración y evita la necesidad de archivos de configuración XML.



Entity Manager: Permite realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en las entidades y administrar el ciclo de vida de los objetos.

Consulta JPQL: Java Persistence Query Language (JPQL) es un lenguaje de consulta orientado a objetos que permite realizar consultas en entidades Java en lugar de tablas de base de datos.

 **Relaciones entre Entidades:** JPA admite la definición de relaciones entre entidades, como uno a uno, uno a muchos y muchos a muchos. Esto se logra mediante el uso de anotaciones como @OneToOne, @OneToMany, y @ManyToMany.

Acceso a datos con JPA

✓ Ventajas:

✕ **Portabilidad:** La capacidad de cambiar fácilmente entre diferentes proveedores de bases de datos sin cambiar el código fuente es una ventaja significativa.

Mapeo Objeto-Relacional: simplifica el mapeo de objetos a tablas de bases de datos, eliminando gran parte del código JDBC manual.

○ **Consulta JPQL:** La capacidad de realizar consultas utilizando JPQL permite a los desarrolladores trabajar con entidades Java en lugar de preocuparse por la estructura de la base de datos, lo que facilita el desarrollo y mantenimiento del código.

— Desventajas:

✕ **Complejidad:** La configuración y el entendimiento completo de JPA pueden requerir tiempo y esfuerzo.

✕ **Rendimiento:** En comparación con consultas SQL personalizadas, las consultas JPQL pueden tener un rendimiento ligeramente inferior en ciertos casos.

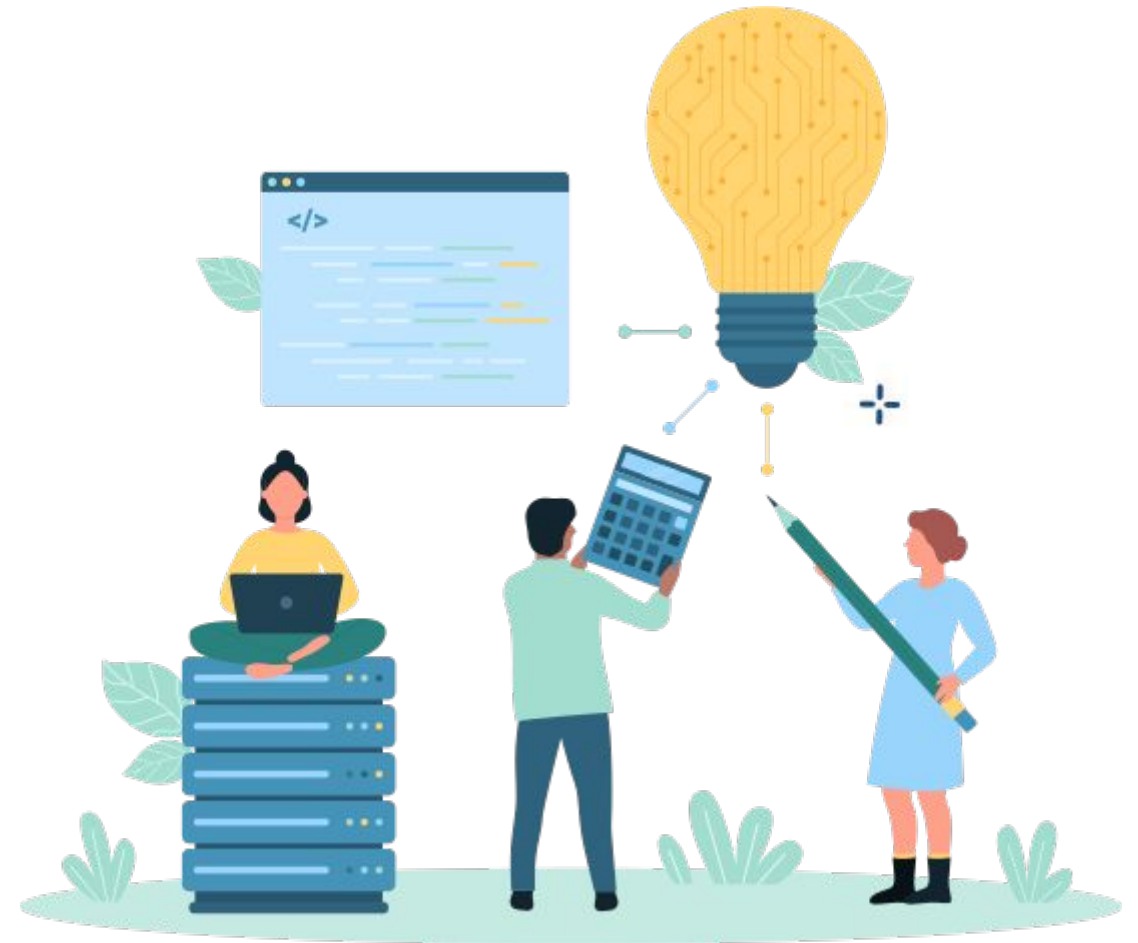
○ **Sobreabstracción:** En algunos casos, la abstracción proporcionada por JPA puede resultar excesiva, especialmente para aplicaciones simples.

› Clases de Entidad



Clases de Entidad

Las clases de entidad son componentes fundamentales que representan objetos persistentes almacenados en una base de datos relacional. Estas clases sirven como puente entre el mundo orientado a objetos de Java y el mundo de las bases de datos relacionales, facilitando el mapeo de datos entre ambas tecnologías.





Clases de Entidad



En otras palabras, los objetos de entidades pueden ser clases concretas o clases abstractas. Podemos decir que **cada Entidad corresponderá con una tabla de nuestra Base de Datos.**



Las clases entidad se identifican con la anotación `@Entity` (a nivel clase).

```
@Entity ←  
public class Usuario {  
    // Atributos, constructores, métodos, etc.  
}
```





Clases de Entidad



Definición de Clase de Entidad:

Una clase de entidad en JPA es una clase de Java ordinaria que se anota con la anotación **@Entity**. **Esta anotación indica a JPA que la clase debe ser persistente**, es decir, que sus instancias pueden ser almacenadas y recuperadas de una base de datos.



```
@Entity ←  
public class Usuario {  
    // Atributos, constructores, métodos, etc.  
}
```






Clases de Entidad



Definición de Clase de Entidad:

Una clase de entidad en JPA es una clase de Java ordinaria que se anota con la anotación **@Entity**. **Esta anotación indica a JPA que la clase debe ser persistente**, es decir, que sus instancias pueden ser almacenadas y recuperadas de una base de datos. 



```
3 import javax.persistence.Entity;
4
5 @Entity
6 public class Usuario {
7     String id;
8     String nombre;
9     String email;
10
```



Clases de Entidad

Identidad y Clave Primaria:



Cada entidad en JPA **debe tener una clave primaria única** que la distinga de otras instancias de la misma entidad. La anotación **@Id** se utiliza para marcar el campo que actuará como clave primaria.



```
6 @Entity
7 public class Usuario {
8
9     @Id
10    String id;
11    String nombre;
12    String email;
13
```



En este ejemplo, el campo **id** se define como **la clave primaria de la entidad Usuario**.



Clases de Entidad

Ventajas de las Entity class:



Reutilización de Código: Al representar objetos persistentes como clases de entidad, se promueve la reutilización de código y la organización lógica de la aplicación.

Facilita el Mapeo Objeto-Relacional: JPA simplifica el mapeo de objetos a tablas de bases de datos a través de las clases de entidad, abstrayendo gran parte de la complejidad asociada con las operaciones de persistencia.



Soporte para Relaciones Complejas: Las clases de entidad permiten modelar relaciones complejas entre objetos, facilitando el diseño de bases de datos relacionales más sofisticadas.

Integración con el Ciclo de Vida: JPA proporciona mecanismos para gestionar el ciclo de vida de las entidades, permitiendo ejecutar lógica personalizada en momentos específicos, como antes o después de persistir una entidad.

Evaluación Integradora ✨

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

Trabajo Integrador del Módulo 💪

Iremos completándolo progresivamente clase a clase.




LIVE CODING

Ejemplo en vivo

Completando la Wallet:

- *Vamos a marcar con la anotación @Entity las clases Usuario y Cuenta.*
- *Además, vamos a marcar los atributos que serán mapeados con la anotación @Id para representar la llave primaria de la tabla.*

 **Tiempo: 10 minutos**



➤ Mapeo con anotaciones



Mapeo con anotaciones

Como ya hemos visto anteriormente, las bases de datos relacionales almacenan la información mediante tablas, filas, y columnas, de manera que para almacenar un objeto en la base de datos, hay que realizar una correlación entre el sistema orientado a objetos de Java y el sistema relacional de nuestra base de datos. JPA nos permite realizar dicha correlación de forma sencilla, realizando por sí misma toda la conversión entre nuestros objetos y las tablas de una base de datos.

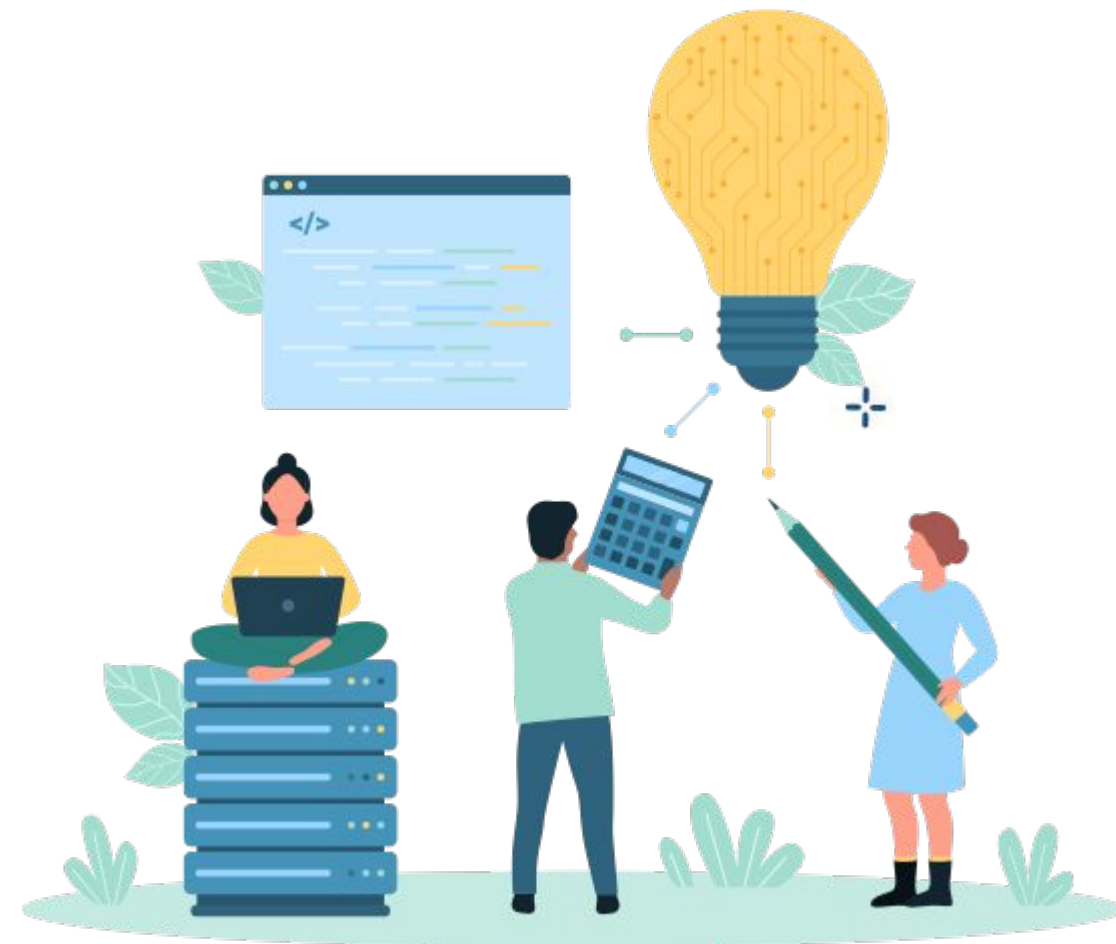




Mapeo con anotaciones

Esta conversión se denomina ORM (Object Relational Mapping - Mapeo Relacional de Objetos), y puede configurarse a través de metadatos (anotaciones).

Las anotaciones nos permiten configurar el mapeo de una entidad dentro del mismo archivo donde se declara la clase, de este modo, relaciona las clases contra las tablas y los atributos contra las columnas. Mediante las anotaciones vamos a explicarle al ORM como debe transformar la entidad en una tabla de base de datos.





Mapeo con anotaciones



Algunas de las principales anotaciones son:

- **@Entity**: Declara la clase como una Entidad.
- **@Table**: Declara el nombre de la Tabla con la que se mapea la Entidad.
- **@Id**: Declara un atributo como la clave primaria de la Tabla.
- **@GeneratedValue**: **Declara la manera en la que el atributo clave primaria va a ser inicializado**. Esto puede ser de forma manual, automática o a partir de una secuencia determinada. La estrategia **IDENTITY** es la más simple de usar, ya que JPA asume que ésta columna es auto generada. Esto provoca que el contador de la columna incremente en 1 cada vez que un nuevo objeto es insertado.





Mapeo con anotaciones

- **@Column:** Declara que un atributo se mapea con una columna de la tabla.
- **@Enumerated:** Declara que un atributo es de alguno de los valores definidos en un Enum (lista de valores constantes). Los valores de un tipo enumerado tienen asociado implícitamente un tipo ordinal.
- **@Temporal:** Declara que se está tratando de un atributo que va a trabajar con fechas, entre paréntesis, debemos especificarle qué estilo de fecha va a manejar en la base de datos:

@Temporal(TemporalType.DATE)

@Temporal(TemporalType.TIME)

@Temporal(TemporalType.TIMESTAMP)



Mapeo con anotaciones

Por ejemplo, para la generación automática de un ID numérico incremental, podríamos utilizar la anotación `@GeneratedValue` de la siguiente manera;

```
8 @Entity
9 public class Cuenta {
10
11     @Id
12     @GeneratedValue(strategy = GenerationType.IDENTITY)
13     Long id;
14
```

La estrategia `IDENTITY` es la que le da el comportamiento incremental.



Maapeo con anotaciones



En cambio, si quisieramos que se genere un id compuesto por una cadena alfanumerica única, utilizamos la anotación **@GeneratedValue** con el generador “**uuid**”



```
7 @Entity
8 public class Usuario {
9
10     @Id
11     @GeneratedValue(generator = "uuid")
12     String id;
```

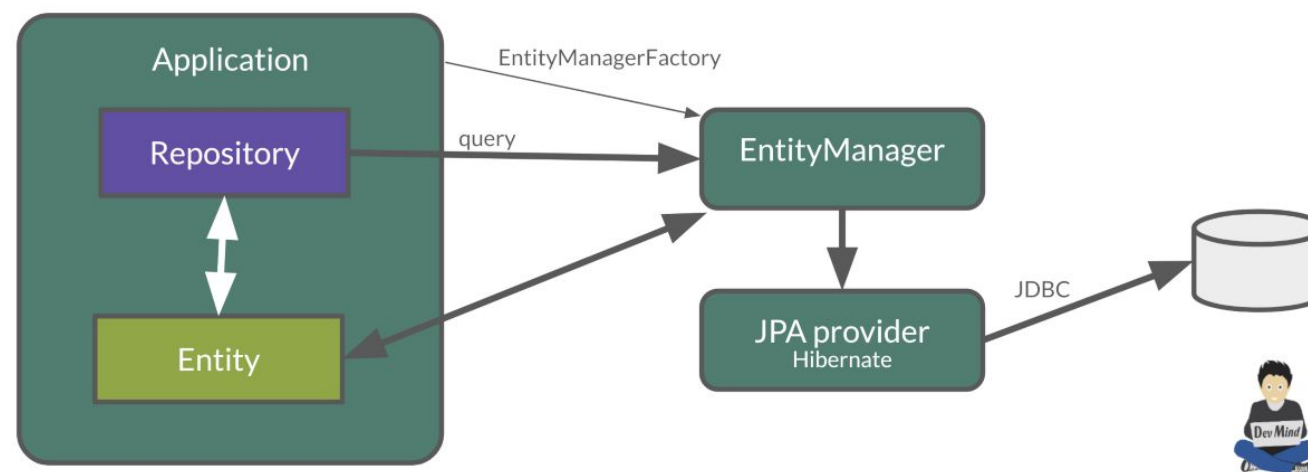


› El EntityManager en JPA

El EntityManager en JPA

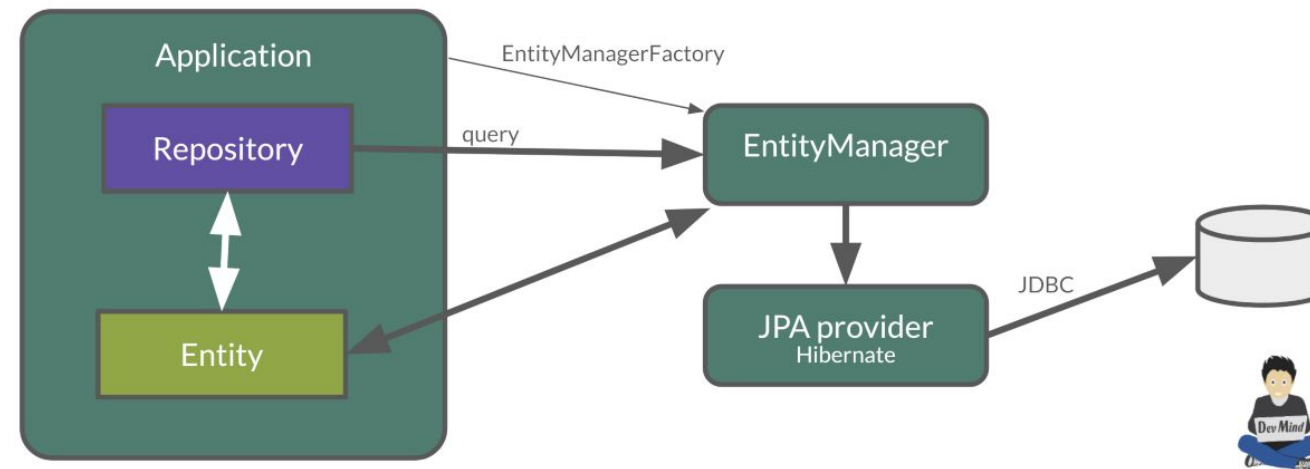
El Entity Manager es un componente central en Java Persistence API (JPA), y su función principal es gestionar el ciclo de vida de las entidades persistentes en una aplicación Java.

Proporciona una interfaz para interactuar con el contexto de persistencia y realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en las entidades mapeadas a una base de datos relacional.



El EntityManager en JPA

El Entity Manager es una interfaz de programación de aplicaciones (API) que forma parte de JPA y se utiliza para gestionar las operaciones de persistencia en entidades. Puede ser considerado como una especie de "puente" entre el mundo de objetos de Java y la base de datos relacional.



El **EntityManager** se obtiene a partir de un **EntityManagerFactory**, que es responsable de crear instancias de Entity Manager.



El EntityManager en JPA

Componentes:



- **Persistence Context:** Este contexto mantiene un conjunto de entidades gestionadas, lo que significa que el Entity Manager está al tanto de los cambios realizados en esas entidades durante la transacción. El Persistence Context asegura la coherencia y la sincronización de los objetos de entidad.
- **Cache de Primer Nivel:** Dentro del Persistence Context, el Entity Manager mantiene una caché de primer nivel que almacena las entidades que han sido recuperadas o gestionadas durante la transacción. Esto mejora el rendimiento al evitar múltiples recuperaciones de la misma entidad.
- **Transacciones:** El Entity Manager se integra estrechamente con el sistema de transacciones de Java, y sus operaciones están típicamente encapsuladas dentro de una transacción. Las transacciones garantizan la consistencia y la atomicidad de las operaciones realizadas en la base de datos.



LIVE CODING

Ejemplo en vivo

Completando la Wallet:

*Ahora vamos a utilizar las anotaciones de generación automática (**@GeneratedValue**) para los ID de las clases entidad:*

- *El **Id** de Usuario tendrá una configuración de generación automática “uuid”.*
- *El **Id** de Cuenta tendrá una configuración de generación automática de tipo Identity.*

Tiempo: 10 minutos



Ejercicio **Entidades**



Manipulación de datos



Contexto: 🙌

Vamos a continuar con el ejercicio de la clase anterior. En este caso vamos a mapear nuestras clases mediante anotaciones JPA.

Consigna: 📝

Anotar una clase con `@Entity`, y anotar con `@Id` el atributo que desees definir como clave primaria de la tabla de la base de datos.

Además, deberás anotar el atributo de clave primaria con alguna generación automática determinada. Si es un String, utiliza el generador “uuid”. Si es un atributo numérico, utiliza la estrategia `IDENTITY`,



Tiempo 🕒: 15 minutos

○

¿Alguna consulta?

+



RESUMEN

¿Qué logramos en esta clase?

- ✓ Conocer la API de Persistencia de Datos de Java (JPA)
- ✓ Comprender sus componentes y principales anotaciones.
- ✓ Aplicar la generación de tablas mediante la anotación @Entity



#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 13 - 16*
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 🙌

