Formación
BDI

Curso Spring Framework

Módulo 6

Acceso a Base de Datos (Persistencia)

spring

Andrés Guzmán F.
Formación BDI TI
Bolsadeideas.com

Temas que veremos

- Soporte a DAO
- Anotación @Repository
- Acceso a datos usando JDBC
  - Clases JdbcTemplate y NamedParameterJdbcTemplate
  - Clases SimpleJdbcTemplate y SimpleJdbcDaoSupport
- Acceso a datos usando ORM
  - Hibernate
  - JPA

Soporte a DAO
(DAO Support)

¿Por qué DAO?

El soporte de Objeto de Acceso a Datos en Spring (en ingles Data Access Object o DAO) está dirigido a proveer consistencia y portabilidad al código, sin importar qué tecnologías de acceso a datos (persistencia) estemos utilizando, ya sea JDBC, Hibernate, JPA o JDO

El cambio de una tecnología de persistencia a otra es una cuestión de cambiar unas pocas líneas en el archivo XML de configuración

También permite a nuestro código no preocuparse por la capturas de excepciones, que son específicas para cada tecnología de persistencia

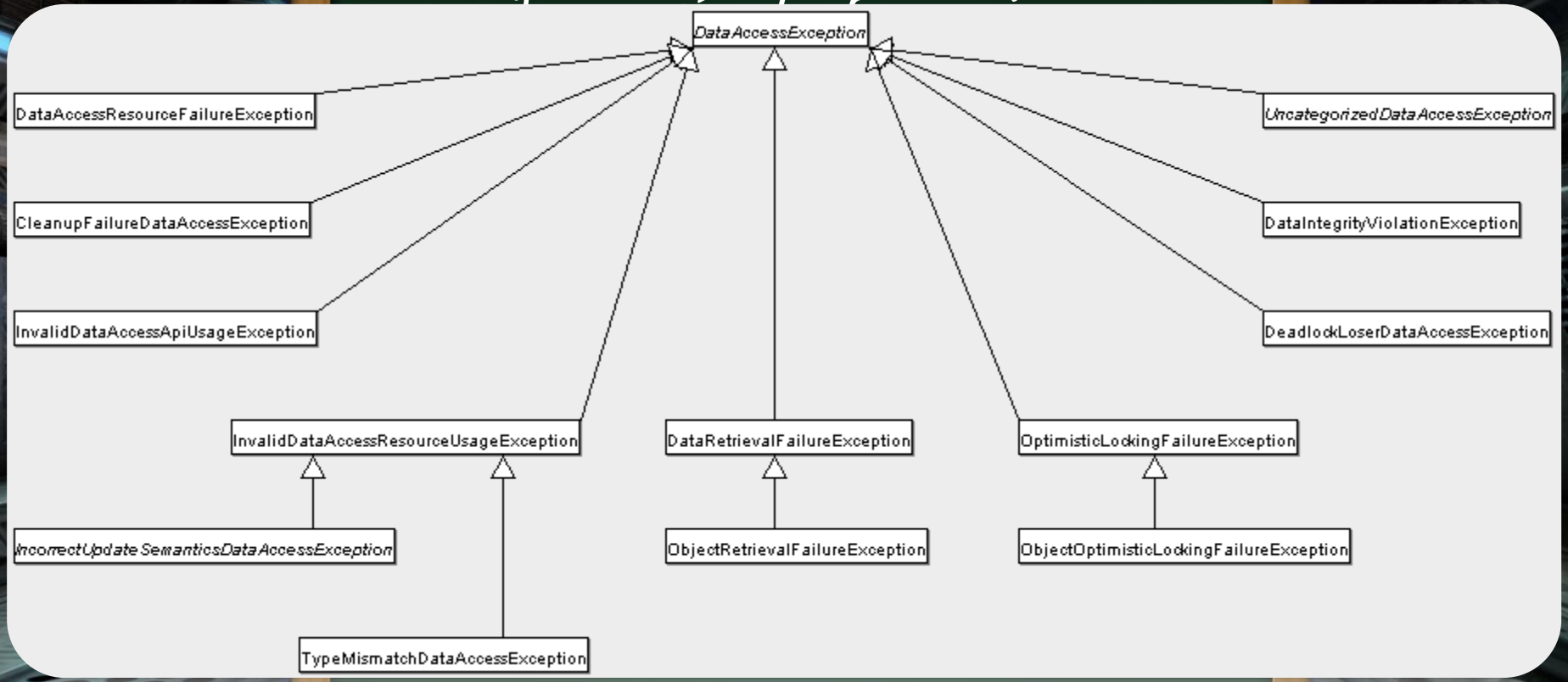
Jerarquía de Excepciones

Spring proporciona muy buena traducción de excepciones de las tecnologías de persistencia como SQLException

Spring maneja su propia jerarquía de clases de excepciones con el DataAccessException como la clase padre

Estas excepciones envuelven la excepción original, de forma simple y en un lenguaje claro, por lo tanto no se corre el riesgo de que se vaya a perder información sobre lo que pudo haber salido mal

Maneja tanto excepciones de JDBC y de Hibernate



Anotación @Repository

Anotación @Repository

Garantiza que nuestros DAOs o repositorios tengan una adecuada traducción de errores/excepciones

Permite el soporte para el escáner de componentes DAOs/Repositorios y que estos puedan ser encontrados y configurados sin tener que ser definidos en el XML de Contexto de Spring

@Repository es un estereotipo de @Component

Anotación @Repository

```
@Repository
public class AlgunDao implements IDao {

// ...

}
```

Acceso a recursos de persistencia

Cualquier implementación de DAO o repositorio tiene que acceder a un recurso de persistencia, dependiendo de la tecnología de persistencia usada:

- Dao basado en JDBC necesita acceder al recurso DataSource JDBC
- Dao basado en Hibernate necesita acceder al SessionFactory
- Dao JPA necesita acceder al EntityManager

La forma más simple de acceder a estos recursos es usar inyección de dependencia para inyectarlos

- @Autowired para inyectar un DataSource JDBC o Hibernate SessionFactory
- @PersistenceContext para inyectar un EntityManager de JPA

Inyectando un DataSource JDBC

Podemos inyectar el DataSource dentro del método encargado de crear/instanciar el objeto JdbcTemplate (clase Spring que provee soporte para trabajar con JDBC) o bien en otra implementación como SimpleJdbcCall

```
@Repository
public class JdbcMovieFinderDao implements MovieFinderDao {

   private JdbcTemplate jdbcTemplate;

   @Autowired
   public void init(DataSource dataSource) {
      this.jdbcTemplate = new JdbcTemplate(dataSource);
   }
// ...
}
```

Inyectando un Hibernate SessionFactory

```
@Repository
public class HibernateMovieFinderDao implements MovieFinderDao {

   private SessionFactory sessionFactory;

   @Autowired
   public void setSessionFactory(SessionFactory sessionFactory) {
      this.sessionFactory = sessionFactory;
   }
// ...
}
```

Inyectando un JPA Entity Manager

```
@Repository
public class JpaMovieFinderDao implements MovieFinderDao {

  @PersistenceContext
   private EntityManager entityManager;

// ...
}
```

Acceso a datos
JDBC

Valor agregado JDBC de Spring

¿Qué hace Spring JDBC? Spring Framework se encarga de todos los detalles de bajo nivel de JDBC:

Acciones	Spring	Tu
Definir parámetros de conexión		X
Abrir conexión	X	
Especiar la sentencia SQL		X
Definir parámetros y proveer sus valores (sentencias preparadas)		X
Preparar y ejecutar sentencias	X	
Se encarga de recorrer el bucle para obtener resultados y convertirlos a objetos (si los hay)	X	
Hacer el trabajo de conversión por cada iteración		X
Procesar excepciones	X	
Manejar transacciones	X	
Cerrar conexión, sentencia y resultados	X	

Opciones para el Acceso JDBC

- Podemos usar JdbcTemplate en dos formas
  - ✓ JdbcTemplate
  - ✓ NamedParameterJdbcTemplate
- SimpleJdbcInsert y SimpleJdbcCall
  - ✓ Optimiza metadatos de base de datos
- MappingSqlQuery, SqlUpdate y StoredProcedure
  - ✓ Más orientado a objetos, similar al diseño JDO Query (Java Data Object)

Clase JdbcTemplate

Clase JdbcTemplate

Encargado de crear y liberar los recursos, lo que ayuda a evitar errores típicos, como olvidarse de cerrar la conexión.

Realiza las tareas básicas de crear sentencias (statement) y su ejecución, dejando a nuestro código proveer el SQL y obtener el resultado.

Ejecuta consultas SQL, sentencias update y llamadas de procedimientos almacenados, realiza iteración sobre ResultSets

Maneja las excepciones JDBC y los traduce a la jerarquía de excepción genérica, más informativas y simples de entender

Ejemplos consultas con JdbcTemplate

```
// Consulta para obtener el número de filas
int rowCount = jdbcTemplate.queryObject("select count(*) from clientes",
Integer.class);


// Consulta utilizando una variable bind:
int clientes = this.jdbcTemplate.queryObject("select count(*) from clientes where
primer_nombre = ?", Integer.class, "Andres");


// Consultando un valor String (apellido)
String apellido = this.jdbcTemplate.queryForObject("select apellido from clientes
            where id = ?", // SQL query a ejecutar
            new Object[]{1234}, // argumentos de la consulta
            String.class); // requiredType – el tipo de dato para el resultado
```

Ejemplo consulta y mapeo con JdbcTemplate

```
// Consulta y poblando un objeto de dominio POJO
// Interface RowMapper es usada por JdbcTemplate para mapear cada registro
// de un ResultSet (fila a fila) a un objeto POJO de dominio.
// sin la necesidad de preocuparse por el manejo de excepciones.
// SQLExceptions serán capturadas y manejadas por JdbcTemplate.
Cliente cliente = this.jdbcTemplate.queryForObject("select primer_nombre,
apellido from clientes where id = ?",
new Object[]{ 7 },
new RowMapper<Cliente>() {
   public Cliente mapRow(ResultSet rs, int rowNum) throws SQLException {
      Cliente cliente = new Cliente();
      cliente.setNombre(rs.getString("primer_nombre"));
      cliente.setApellido(rs.getString("apellido"));
      return cliente;
   }
});
```

Ejemplo execute con JdbcTemplate

```
// Usamos método execute(..) para ejecutar un SQL, en especial
// para sentencias DDL y llamadas a procedimiento almacenado.
// Ejecutar una sentencias DDL
this.jdbcTemplate.execute("create table clientes (id integer, nombre
varchar(100))");

// Invocar un procedimiento almacenado
this.jdbcTemplate.execute("call SUPPORT.REFRESH_CLIENTS_SUMMARY(?)",
Long.valueOf(unionId));
```

Clase NamedParameterJdbcTemplate

NamedParameterJdbcTemplate

La clase NamedParameterJdbcTemplate agrega soporte para implementar sentencias JDBC con parámetros en base a nombres (más flexible), sin tener que preocuparnos por el orden de los parámetros, en contraposición a las sentencias JDBC que utilizan los clásico argumentos placeholder ('?')

La clase NamedParameterJdbcTemplate encapsula por debajo a JdbcTemplate y delega gran parte del trabajo a JdbcTemplate

Interface SqlParameterSource

Contiene las asignaciones de valores de los parámetros npmbrados de la sentencia NamedParameterJdbcTemplate

Implementaciones

- MapSqlParameterSource
- BeanPropertySqlParameterSource: envuelve a un JavaBean, utilizando los atributos del objeto como la fuente para los valores de los parámetro nombrados

NamedParameterJdbcTemplate

```
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {
   this.namedParameterJdbcTemplate = new
NamedParameterJdbcTemplate(dataSource);
}

public int cantidadClientesPorNombre(String nombre) {
   String sql = "select count(*) from T_CLIENTES where nombre = :nombre";
   SqlParameterSource namedParameters =
                                    new MapSqlParameterSource("nombre", nombre);

   return namedParameterJdbcTemplate.queryForObject(sql, namedParameters,
String.class);
}
```

JdbcDaoSupport

Estilo clásico JdbcTemplate

```
// Estilo Clasico JdbcTemplate...
// Aquí se muestra como una comparación con el SimpleJdbcTemplate
private JdbcTemplate jdbcTemplate;

public void setDataSource(DataSource dataSource) {
   this.jdbcTemplate = new JdbcTemplate(dataSource);
}

public Cliente findCliente(String especialidad, int edad) {
   String sql = "select id, primer_nombre, apellido from clientes"
                    + " where especialidad = ? and edad = ?";

   RowMapper<Cliente> mapper = new RowMapper<Cliente>() {
         public Cliente mapRow(ResultSet rs, int rowNum) throws SQLException {
         Cliente cliente = new Cliente();
         cliente.setId(rs.getLong("id"));
         cliente.setNombre(rs.getString("primer_nombre"));
         cliente.setApellido(rs.getString("apellido"));
         return cliente;
      }};

   // note que envolvemos los valores en un arreglo
   return this.jdbcTemplate.queryForObject(sql, new Object[]{especialidad, edad},
mapper);
}
```

JdbcDaoSupport

Extensión de JdbcDaoSupport envuelve internamente a JdbcTemplate para dar soporte DAOs

JdbcDaoSupport y NamedParameterJdbcDaoSupport dan soporte mucho más completo, robusto, pero a la vez simple, para implementar beans de acceso a datos o DAOs, proveen una plantilla que se encargan transparentemente de las operaciones típicas como consultas update, insert, delete

Ejemplo JdbcDaoSupport

```
public class JdbcEstudianteDao extends JdbcDaoSupport
                                    implements EstudianteDao {

 public void save(Estudiante estudiante) {

     String sql = "INSERT INTO ESTUDIANTE (ID, NOMBRE, FECHA, PROMEDIO) "
                                    + "VALUES (?, ?, ?, ?)";
     getJdbcTemplate().update(sql,
          new Object[] {estudiante.getId(), estudiante.getNombre(),
               estudiante.getFechaNacimiento(),
               estudiante.getPromedio() });
   }
...
}
```

Ejemplo NamedParameterJdbcDaoSupport

```
public class JdbcEstudianteDao extends NamedParameterJdbcDaoSupport
                                    implements EstudianteDao {

...

   // Usa parámetros nombrados - más flexible
   public void update(Estudiante estudiante) {
      String sql = "UPDATE ESTUDIANTE SET
                              NOMBRE = :nombre,
                              PROMEDIO = :promedio,
                              FECHA_NACIMIENTO = :fechaNacimiento
                     WHERE ID = :id"";


      SqlParameterSource parameterSource = new
                                    BeanPropertySqlParameterSource(estudiante);
     getNamedParameterJdbcTemplate().update(sql, parameterSource);
   }
...
}
```

Soporte ORM en Spring

Soporte ORM en Spring

Spring soporta administración de recursos, implementación data access object (DAO) y control de transacción

- Hibernate
- JPA (Java Persistence API)
- JDO
- iBATIS SQL Maps

Beneficios de usar DAOs de Spring

- Fácil de testear
- Excepciones de acceso a datos más entendibles
- Mejor administración de recursos
- Fácil configuración del Hibernate SessionFactory y JPA EntityManagerFactory en XML de Spring (archivo root/application contexts)

Beneficios de usar DAOs de Spring

Integrado con transaction management (manejo de transacciones)

- Podemos envolver nuestro código ORM (DAOs) con una declarativa programación orientada a aspectos (AOP) ya sea anotando un método con @Transactional el cual será interceptado por el aspecto para aplicar la transacción, o bien configurando el advice AOP en el XML root/application contexts de spring

Dependencia Maven

```
<!-- Spring ORM -->
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-orm</artifactId>
   <version>4.2.0.RELEASE</version>
</dependency>
```

Porqué el Mapeo de Objeto Relacional (ORM)?

- La capa de persistencia es una importante parte de cualquier proyecto de desarrollo de aplicaciones empresariales
  - ✓ Accede y opera con los datos persistentes típicamente de una base de datos relacional
- Las Bases de datos relacionales manejan tablas (con filas y columnas)
  - Nosotros, los desarrolladores Java, queremos trabajar con clases/objetos, no filas y columnas
  - ORM se encarga del mapeo entre los dos

Porqué el Mapeo de Objeto Relacional (ORM)?

ORM maneja relaciones de objetos

Diseñado con un excelente rendimiento, rápidas consultas/operaciones en la BBDD, maneja dos niveles de cache, donde el primer nivel es automático

El mapeo objeto-relacional (Object-Relational mapping, o sus siglas ORM) es una arquitectura que permite trabajar con los datos de una base de datos relacional (RDBMS) en forma de objetos (lenguaje POO)

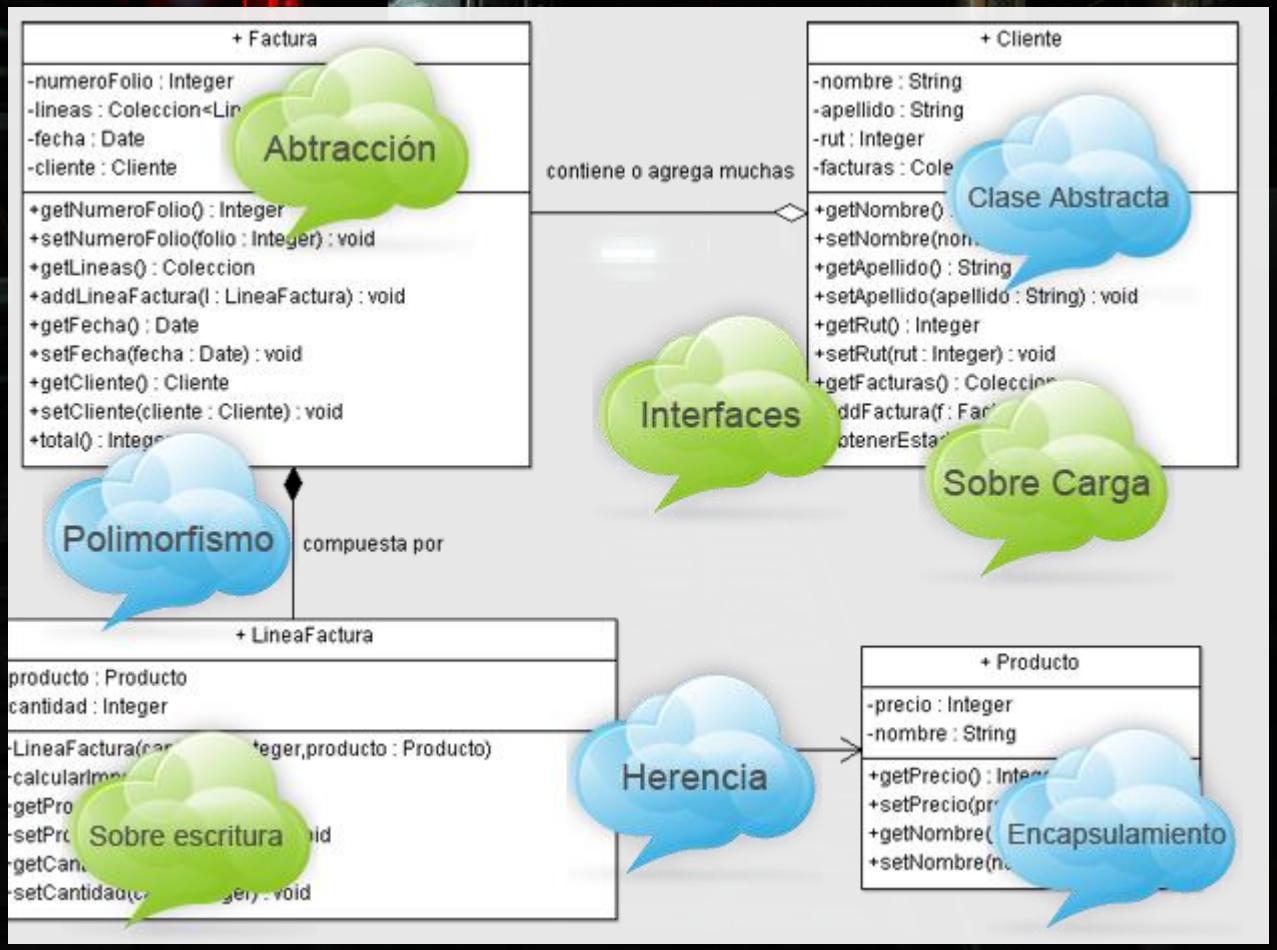
Esto crea una base de datos orientada a objetos en nuestra aplicación mapeada a una base de datos relacional, es decir una base de datos virtual de objetos, sobre la base de datos relacional

- ✓ Esto funciona asociando a cada tabla de la base de datos un Plain Old Java Object (POJO o clase Entity)
- ✓ Cada atributo de la clase Entity es mapeado o asignado a las columnas de la tabla de la base de datos
- ✓ Un Entity es similar a una Java Bean, con propiedades accesibles mediante métodos setter y getter

id	cliente_id	solicitud_id	status_id	created_at	updated_at	numero_operacion	plazo_entrega	total	descripcion
1	6	1	4	2014-02-16 20:38:32	2014-02-16 20:38:55	VE10001	2014-02-16 20:38:55	8889	alguna nota

id	factura_id	producto_id	cantidad	importe
2	1	1	4	8888

Esto permite el uso de las características propias de la programación orientada a objetos (relaciones, herencia y polimorfismo)



Clases de Dominio o Entidades

Clases de Dominio o Entity

- Las clases de dominio son aquellas en una aplicación que implementan las entidades del dominio de negocio (por ejemplo, Cliente y OrdenCompra en una aplicación de E-commerce)

- Un ORM como Hibernate o JPA2 funciona mejor si estas clases siguen algunas reglas simples, también conocido como el modelo de programación Plain Old Java Object (POJO).

Clases de Dominio o Entity

Objetos Mapeados

Objetos Manejados

Objetos Anotados

Objetos Persistentes

Objetos De Consultas

Objetos Cacheados

Los objetos son instancias que solo viven en memoria

Los entity son objetos que viven en modo persistente en la Base de Datos

Una entidad es un objeto de persistencia que representa una tabla en el modelo de datos relacional y cada instancia de esta entidad corresponde a un registro en esa tabla

El estado de persistencia de una entidad se representa a través de atributos persistentes

Estos atributos usan XML o Anotaciones para el mapeo de estos objetos en el modelo de base de datos

Usamos XML o Anotaciones de Hibernate para mapear las clases de dominio de persistencia

Pasos para escribir una clase Entity

Paso 1: Implementar constructor sin argumentos

- Toda clase persistente deben tener un constructor por defecto de manera que Hibernate pueda crear instancias de ellas

Paso 2: Proveer un atributo Id (identificador)

- Este atributo se asigna (mapeada) al campo de llave primaria de una tabla de base de datos.
- Este atributo puede ser nombrado como se quiera, y el tipo de dato puede ser cualquier primitivo java.lang.String o java.lang.Integer

Pasos para escribir una clase Entity

Paso 3: Declarar los métodos getter/setter para los atributos persistentes

Paso 4: Implementar la interfaz java.io.Serializable

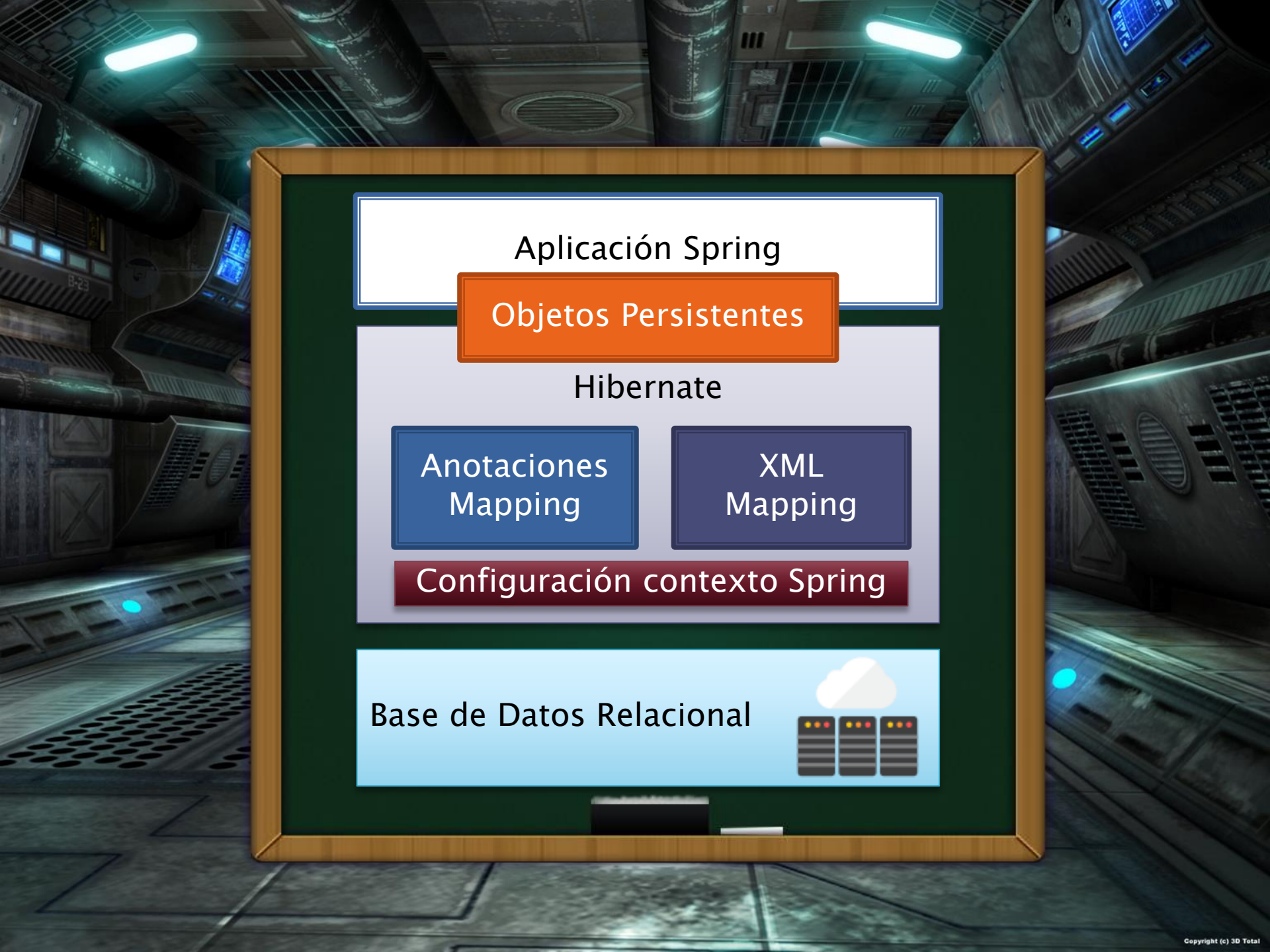
Hibernate

¿Qué es Hibernate?

Hibernate es una herramienta de Mapeo objeto-relacional (ORM) para la plataforma Java, es el Framework más popular de Persistencia que habilita la persistencia en clases POJO de forma transparente.

Facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante objetos persistentes que siguen fielmente los conceptos y paradigmas comunes de la programación orientada a objetos

Utiliza archivos declarativos (XML) o anotaciones en las clases de las entidades que permiten establecer estas relaciones.


Aplicación Spring
Objetos Persistentes
Hibernate
Anotaciones Mapping
XML Mapping
Configuración contexto Spring
Base de Datos Relacional

Dos formas para mapear una clase POJO de Dominio

Usando archivo XML

- Describen el mapeo relacional entre una clase POJO de dominio (o entity) y una tabla de la base de datos
- Estudiantes.hbm.xml

Usando anotaciones JPA

- Usando anotación @Entity sobre las clases de dominio o entity
- No necesita ningún tipo de xml

Dependencia Maven

```
<!-- Hibernate -->
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-core</artifactId>
   <version>4.3.11.Final</version>
</dependency>
```

Forma #1: Mapeo usando XML

Configurando Mapeos con XML en el archivo XML de Spring Hibernate Context

```
<beans>
...
<!-- Hibernate SessionFactory -->
<bean id="sessionFactory"
class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
   <property name="dataSource" ref="dataSource" />
   <property name="mappingResources">
      <list>
         <value>Estudiante.hbm.xml</value>
      </list>
   </property>
   <property name="hibernateProperties">
      <props>
         <prop key="hibernate.dialect">org.hibernate.dialect.DerbyDialect</prop>
         <prop key="hibernate.show_sql">true</prop>
      </props>
   </property>
</bean>
</beans>
```

Archivo XML de mapeo de Hibernate: Estudiante.hbm.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="com.formacionbdi.ejemplo.entity">
   <class name="Estudiante" table="T_ESTUDIANTES">

      <id name="id" type="int" column="ID">
         <generator class="identity"/>
      </id>

      <property name="nombre" type="string" />

      <property name="fechaNacimiento" type="date"
                  column="fecha_nacimiento"/>

      <property name="promedio" type="int" column="promedio"/>

   </class>
</hibernate-mapping>
```

Asignación de clases a las tablas

- El mapeo de clases a las tablas se define con el elemento XML class

```
<class name="model.entity.Producto" table="tbl_productos">
...
</class>
```

Atributos del elemento <class>

```
<class
    name="ClassName"                                  (1)
    table="tableName"                                 (2)
    discriminator-value="discriminator_value“         (3)
    mutable="true|false"                              (4)
    schema="owner"                                    (5)
    catalog="catalog"                                 (6)
    proxy="ProxyInterface"                            (7)
    dynamic-update="true|false"                       (8)
    dynamic-insert="true|false"                       (9)
    select-before-update="true|false“                 (10)
    polymorphism="implicit|explicit"                  (11)
    where="arbitrary sql where condition"             (12)
    persister="PersisterClass"                        (13)
    batch-size="N"                                    (14)
    optimistic-lock="none|version|dirty|all"          (15)
    lazy="true|false"                                 (16)
    entity-name="EntityName"                          (17)
    check="arbitrary sql check condition"             (18)
    rowid="rowid"                                     (19)
    subselect="SQL expression"                        (20)
    abstract="true|false"                             (21)
    node="element-name"
/>
```

Atributos del elemento <class>

- **name** Corresponde al nombre completo de la clase Java persistente (incluyendo su package)
- **table** (opcional, por defecto es el nombre de la clase): El nombre de la tabla de base de datos correspondiente
- **discriminator-value** (opcional, por defecto es el nombre de la clase): Un valor que distingue subclases concretas, usado para el comportamiento de herencia y polimorfismo

Atributos del elemento <class>

- **dynamic-update** (opcional, por defecto es false): Especifica que SQL UPDATE debe ser generado en tiempo de ejecución y contiene sólo aquellas columnas cuyos valores han cambiado
- **dynamic-insert** (opcional, por defecto es false): Especifica que SQL INSERT debe ser generado en tiempo de ejecución y contiene sólo las columnas cuyos valores no son nulos
- **optimistic-lock** (opcional, por defecto es version): Determina la estrategia del optimistic locking

Asignación de atributos del objeto persistente a las columnas

- El mapeo de atributos de las clases entity a las columnas de las tablas DDBB se define con el elemento XML property

```
<property
   name="reason"
   type="java.lang.String"
   update="true"
   insert="true"
   column="reason"
   not-nul="true" />
```

Atributos del elemento <property>

```
<property
    name="propertyName"
    column="column_name"
    type="typename"
    update="true|false"
    insert="true|false"
    formula="arbitrary SQL expression"
    access="field|property|ClassName"
    lazy="true|false"
    unique="true|false"
    not-null="true|false"
    optimistic-lock="true|false"
    generated="never|insert|always"
    node="element-name|@attribute-name|element/@attribute|."
    index="index_name"
    unique_key="unique_key_id"
    length="L"
    precision="P"
    scale="S"
/>
```

Atributos del elemento <property>

- **name** el nombre del atributo o la propiedad de la clase persistente, con una letra minúscula inicial
- **column** (opcional, por defecto lleva el mismo nombre del atributo de la clase): corresponde al nombre de la columna de la tabla de la base de datos asignada, cuando se llama igual que el atributo es opcional
- **unique** (opcional): Activa el DDL de una restricción de unicidad para la columna
- **not-null** (opcional): Activa la generación DDL de una restricción nulabilidad para la columna

Asignación de la llave primaria o campo Id

- Usa el elemento id para el mapeo
- Usa el sub-elemento generator con el atributo class, que especifica el esquema de generación de llaves

```
<class name="model.entity.Persona">
   <id name="id" type="int">
      <generator class="increment"/>
   </id>
  <property name="nombre" column="cname" type="string"/>
</class>
```

Esquema de generación de claves

El esquema de generación de claves se configura a través del atributo class:

- **class="increment"**
  - ✓ Genera identificadores del tipo long, short o int, son únicos sólo cuando ningún otro proceso está insertando datos en la misma tabla. No debe ser utilizado en entorno de clúster
- **class="identity"**
  - ✓ Soportado para llaves primarias/ columnas identity en DB2, MySQL, MS SQL Server, Sybase y HypersonicSQL

Esquema de generación de claves

- **class="hilo"**
  - ✓ Utiliza algoritmo hi/lo para generar eficientemente llaves del tipo long, short o int, dada una tabla y columna
- ✓ **class="assigned"**
  - ✓ Permite a la aplicación asignar un identificador (manualmente vía setter o constructor) al objeto antes de que sea guardado con el método save(...)
  - ✓ Es la estrategia por defecto si no se especifica un elemento <generator>

Esquema de generación de claves

- **class="native"**
  - ✓ Obtiene una identidad, secuencia o hilo dependiendo de las capacidades de la base de datos de origen

- **class="uuid"**
  - ✓ Utiliza un algoritmo UUID de 128 bits para generar llaves

Forma #2: Mapeo usando Anotaciones

Configurando Mapeos con Anotaciones en el archivo XML de Spring Hibernate Context

```
<beans>
...
<!-- Hibernate SessionFactory -->
<bean id="sessionFactory"
class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
   <property name="dataSource" ref="dataSource" />
   <property name="annotatedClasses">
      <list>
         <value>com.formacionbdi.ejemplo.entity.Estudiante</value>
      </list>
   </property>
   <property name="hibernateProperties">
      <props>
         <prop key="hibernate.dialect">org.hibernate.dialect.DerbyDialect</prop>
         <prop key="hibernate.show_sql">true</prop>
      </props>
   </property>
</bean>
</beans>
```

Usando la anotación @Entity sobre la clase de dominio o entity

```
@Entity
@Table(name = "T_ESTUDIANTES")
public class Estudiante implements Serializable {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(name = "id")
   private Integer id;

   @Column(name = "nombre", length = 20, nullable = false)
   private String nombre;

   @Column(name = "fecha_nacimiento")
   private Date fechaNacimiento;

   @Column(name = "promedio")
   private int promedio;

   public Estudiante() {}

   public Student(String nombre, Date fechaNacimiento, int promedio) {
      this.nombre = nombre;
      this.fechaNacimiento = fechaNacimiento;
      this.promedio = promedio;
   }
....
```

¿Cómo hace Hibernate para mapear Usando anotaciones?

Hibernate JPA asigna objetos a una base de datos mediante el uso de metadatos

Las Entidades tienen asociados metadatos que describen el mapeado

Estos metadatos son anotaciones descritas en javax.persistence

Hibernate JPA cuenta con reglas de mapeados por defecto, solo necesitamos anotaciones para excepciones, configuración por excepciones

Mapeo de una clase Entidad

```
@Entity
@Table(name = "estudiantes")
public class Estudiante {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Integer id;

    @Column(length = 50, nullable = false)
    private String nombre;

    @Column(name = "fecha_nacimiento")
    private Date fechaNacimiento;

    private int nota;

    public Estudiante() {
    }

    public Estudiante(String nombre, Date fechaNacimiento, int nota){
        this.nombre = nombre;
        this.fechaNacimiento = fechaNacimiento;
        this.nota = nota;
    }
...etc... Getters y setters
```

Para ser reconocida como una Entidad la clase debe aparecer con la anotación **@Entity**

La anotación **@Id** denota la lcave primaria, su valor es generado por **@GeneratedValue**

@Column usado en atributos para sobrescribir el mapeado de columnas por defecto

Asigna Entity Estudiante a la tabla estudiantes
Genera una clave primaria incremental
Sincroniza valores de atributos en columnas de la BD

Mapeo de una clase Entidad

- **@Entity:** Indica que es una clase POJO de Entidad (representa una tabla en la base de datos)

- **@Table:** Específica la tabla principal relacionada con la entidad.
  - name - nombre de la tabla, por defecto el de la entidad si no se especifica
  - catalog - nombre del catálogo
  - schema - nombre del esquema

Mapeo de una clase Entidad

- **@Id:** Indica la clave primaria de la tabla
- **@GeneratedValue:** Asociado con la clave primaria, indica que ésta se debe generar por ejemplo con una secuencia de la base de datos
  - strategy - estrategia a seguir para la generación de la clave: AUTO (valor por defecto, el contenedor decide la estrategia en función de la base de datos), IDENTITY (utiliza un contador, ej: MySQL), SEQUENCE (utiliza una secuencia, ej: Oracle, PostgreSQL) y TABLE (utiliza una tabla de identificadores).
  - generator - forma en la que genera la clave.

Mapeo de una clase Entidad

- **@Column:** Especifica una columna de la tabla a mapear con un campo de la entidad
  - name - nombre de la columna.
  - unique - si el campo es único
  - nullable - si permite nulos.
  - insertable - si la columna se incluirá en la sentencia INSERT
    updatable - si la columna se incluirá en la sentencia UPDATE
  - length - longitud de la columna.
  - precision - número de decimales
  - scale - escala decimal

¿Qué es un DAO?
Preview
Data Access Object

¿Qué es un DAO?

En Ingeniería de software, un Data Access Object (DAO) es una clase que implementa y provee una interfaz común para acceder y trabajar con los datos, independiente de las tecnologías a utilizar JDBC, JPA, Hibernate, TopLink, OpenJpa, Codo, EclipseLink, iBATIS o JDO

Esta interface tiene que tener los métodos necesarios para recuperar y almacenar los datos (contrato de implementación) con las operaciones básicas: listar, obtener por id, guardar, eliminar etc

¿Qué es un DAO?

Separación de la capa de acceso a datos/lógica de negocio (persistencia). Permite una fácil sustitución de la base de datos sin afectar a la lógica de negocio

Entonces resumiendo un DAO es simplemente una clase del modelo que se encarga de implementar las operaciones básicas de acceso a datos que debe implementar un contrato, la interfaz.

Para implementar nuestros DAO usaremos el componente Session que forma parte del API de Hibernate, o bien, si usamos JPA utilizamos el Entity Manager.

Data Access Object (DAO)

Se encargan de las tareas cotidianas detrás de escena por nosotros,

por ejemplo consultas y aplicaciones CRUD (Crear, Obtener, Actualizar y Borrar) de manera muy simple,

con muy poco código, todo está mecanizado

y automatizado por debajo

Data Access Object (DAO)

También se encarga de realizar y ejecuta las consultas del tipo SELECT (ORM HQL- Hibernate Query Language): crear el objeto, prepara las consultas y ejecutar

Implementar DAO con Hibernate

Dos opciones para implementar el Modelo (DAOs):

Opción #1: Usando SessionFactory

- SessionFactory forma propia y nativa de Hibernate con sus propias excepciones para implementar DAOs

Opción #2: Usando HibernateTemplate

- HibernateTemplate es una clase helper de spring que simplifica el trabajo con hibernate y el acceso a los datos
- Convierte automáticamente las excepciones de hibernate HibernateExceptions ien DataAccessExceptions, siguiendo la jerarquía de excepción de spring (org.springframework.dao)
- Recomendado sobre la opción #1

Opción #1: Usando SessionFactory

```
@Repository("estudianteDao")
public class HibernateEstudianteDao implements EstudianteDao {

   @Autowired
   private SessionFactory sessionFactory;

   public void setSessionFactory(SessionFactory sessionFactory) {
      this.sessionFactory = sessionFactory;
   }

   @Transactional
   public void save(Estudiante estudiante) {
      sessionFactory.getCurrentSession().saveOrUpdate(estudiante);
   }

   @Transactional
   public void delete(Integer estudianteId) {
      Estudiante estudiante = (Estudiante) sessionFactory.getCurrentSession().
                                                 get(Estudiante.class, estudianteId);
      sessionFactory.getCurrentSession().delete(estudiante);
   }

   @Transactional(readOnly = true)
   public Estudiante findById(Integer estudianteId) {
      return (Estudiante) sessionFactory.getCurrentSession().
                                                 get(Estudiante.class, estudianteId);
   }
}
```

Opción #2: Usando HibernateTemplate

```
@Repository("estudianteDao")
public class HibernateEstudianteDao implements EstudianteDao {

   @Autowired
   private HibernateTemplate hibernateTemplate;

   public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {
      this.hibernateTemplate = hibernateTemplate;
   }

   @Transactional
   public void save(Estudiante estudiante) {
      hibernateTemplate.saveOrUpdate(estudiante);
   }

   @Transactional
   public void delete(Integer estudianteId) {
      Estudiante estudiante = (Estudiante) hibernateTemplate
                                                .get(Estudiante.class, estudianteId);
      hibernateTemplate.delete(estudiante);
   }

   @Transactional(readOnly = true)
   public Estudiante findById(Integer estudianteId) {
      return (Estudiante) hibernateTemplate.get(Estudiante.class, estudianteId);
   }
}
```

JPA

Java Persistence API, más conocida por su sigla JPA, es la API de persistencia desarrollada para la plataforma Java EE e incluida en el estándar EJB3

Busca unificar las diferentes tecnologías que proveen un mapeo objeto-relacional bajo un único estándar

El objetivo del diseño de esta API es no perder de vista las ventajas de la orientación a objetos al interactuar con una base de datos

y usar objetos simples (conocidos como Entity o POJOs).

Java Persistence API

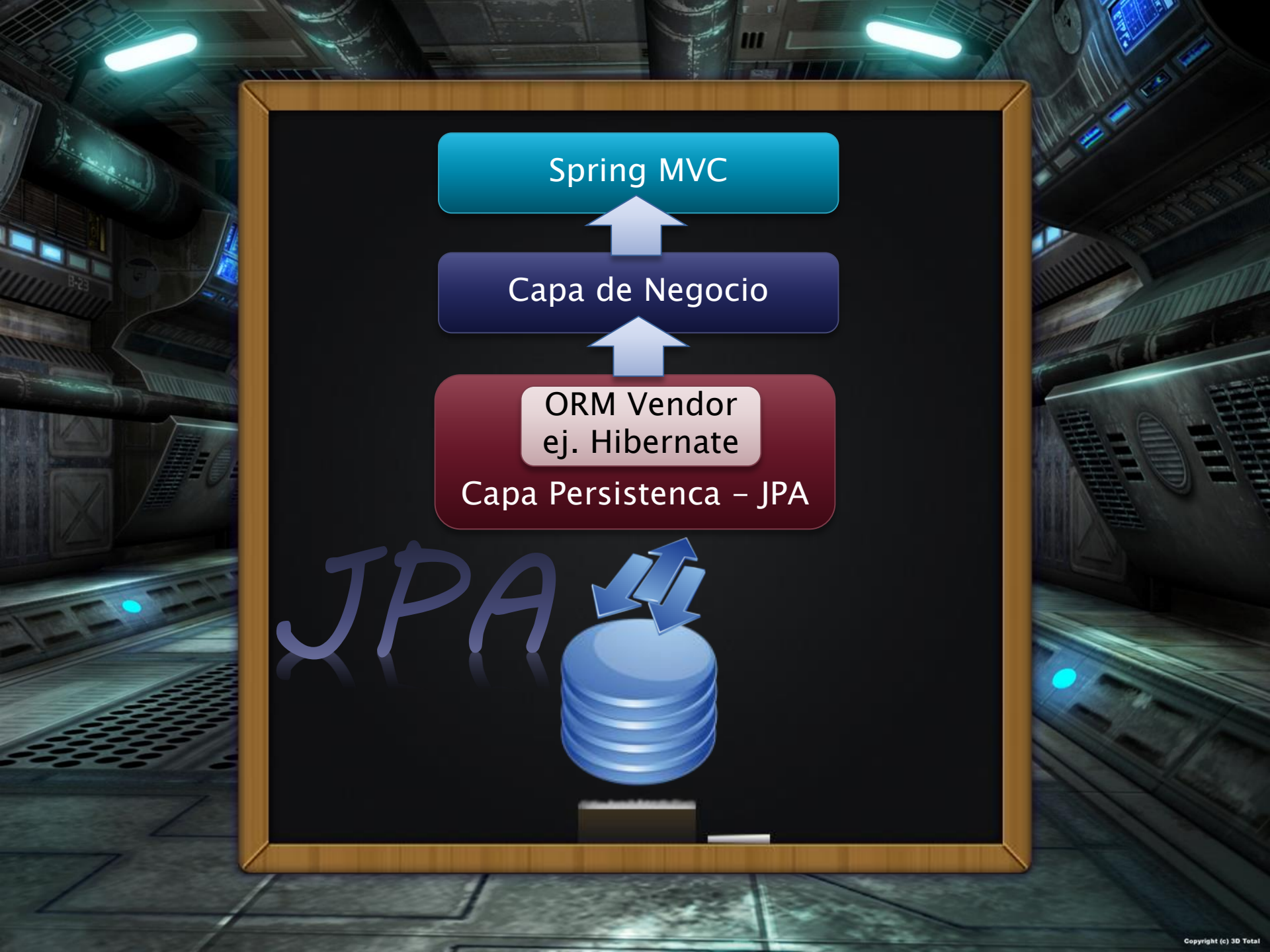
Para trabajar con la especificación JPA en spring, primero tenemos que seleccionar un proveedor o implementación de JPA

Hibernate es la implementación preferida

Entre las posibles opciones tenemos

JPA

- Hibernate
- OpenJPA
- EclipseLink


Spring MVC
Capa de Negocio
ORM Vendor
ej. Hibernate
Capa Persistenca - JPA
JPA

Seleccionar un proveedor JPA

En JPA, debemos seleccionar un proveedor o implementación en la configuración de contexto de spring

Por ejemplo, podemos escoger entre los siguientes proveedores JPA

- Hibernate
- OpenJPA
- EclipseLink

Configurando JPA y seleccionando un proveedor

```
<!-- JPA EntityManagerFactory -->
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
   <property name="dataSource" ref="dataSource" />
   <property name="jpaVendorAdapter">

      <!-- Usando Hibernate como proveedor JPA -->
      <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"
               p:database="HSQL"
               p:showSql="true" />

      <!-- Usando OpenJPA como proveedor JPA, comentado para usar Hibernate -->
      <!-- <bean class="org.springframework.orm.jpa.vendor.OpenJpaVendorAdapter"
                  p:database="HSQL"
                  p:showSql="true"/> -->
   </property>
   <property name="persistenceXmlLocation" value="classpath:META-INF/persistence.xml" />
</bean>
```

Tres opciones para configurar JPA

Spring soporta tres formas de configurar JPA EntityManagerFactory que seran utilizada por la aplicación para obtener un entity manager. Cada forma es para un contexto de servidor y tipos de aplicaciones diferentes

- LocalEntityManagerFactoryBean: sólo para aplicaciones de consola o escritorio, o para desarrollo o testing.
- Obtener un EntityManagerFactory desde JNDI (solo si usamos algún Servidor de Aplicaciones Java EE como JBOSS AS o Glassfish)
- LocalContainerEntityManagerFactoryBean (Recomendada para Spring Web MVC)

LocalEntityManagerFactoryBean

- Sólo use esta opción en entornos de despliegue simples como aplicaciones de consola/escritorio o pruebas
- Esta forma de implementación JPA es la más simple y limitada de todas
- No se puede hacer referencia a una definición de bean JDBC DataSource y no tiene soporte para transacciones globales

```
<beans>
  <bean id="entityManagerFactory"
          class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
    <property name="persistenceUnitName" value="algunaUnidadPersistencia"/>
  </bean>
</beans>
```

Obtener EntityManagerFactory desde JNDI

Utilice esta opción cuando se despliegue en un Servidor de Aplicaciones Java EE, por ejemplo JBOSS AS o Glassfish

```
<beans>
   <jee:jndi-lookup id="entityManagerFactory"
                    jndi-name="persistence/algunaUnidadPersistencia"/>
</beans>
```

LocalEntityManagerFactoryBean

Utilice esta opción para entornos de aplicaciones Spring Web MVC, con todas las características disponibles.

- Permite configuración fina y detallada
- Seleccionar un motor o proveedor JPA entre otras cosas

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
   <property name="dataSource" ref="dataSource" />
   <property name="jpaVendorAdapter">
      <bean
          class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"
               p:database="HSQL"
               p:showSql="true" />
   </property>
   <property name="persistenceXmlLocation" value="classpath:META-
INF/persistence.xml" />
</bean>
```

Ejemplo del archivo persistence.xml

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0">

  <persistence-unit name="algunaUnidadPersistencia"/>
</persistence>
```

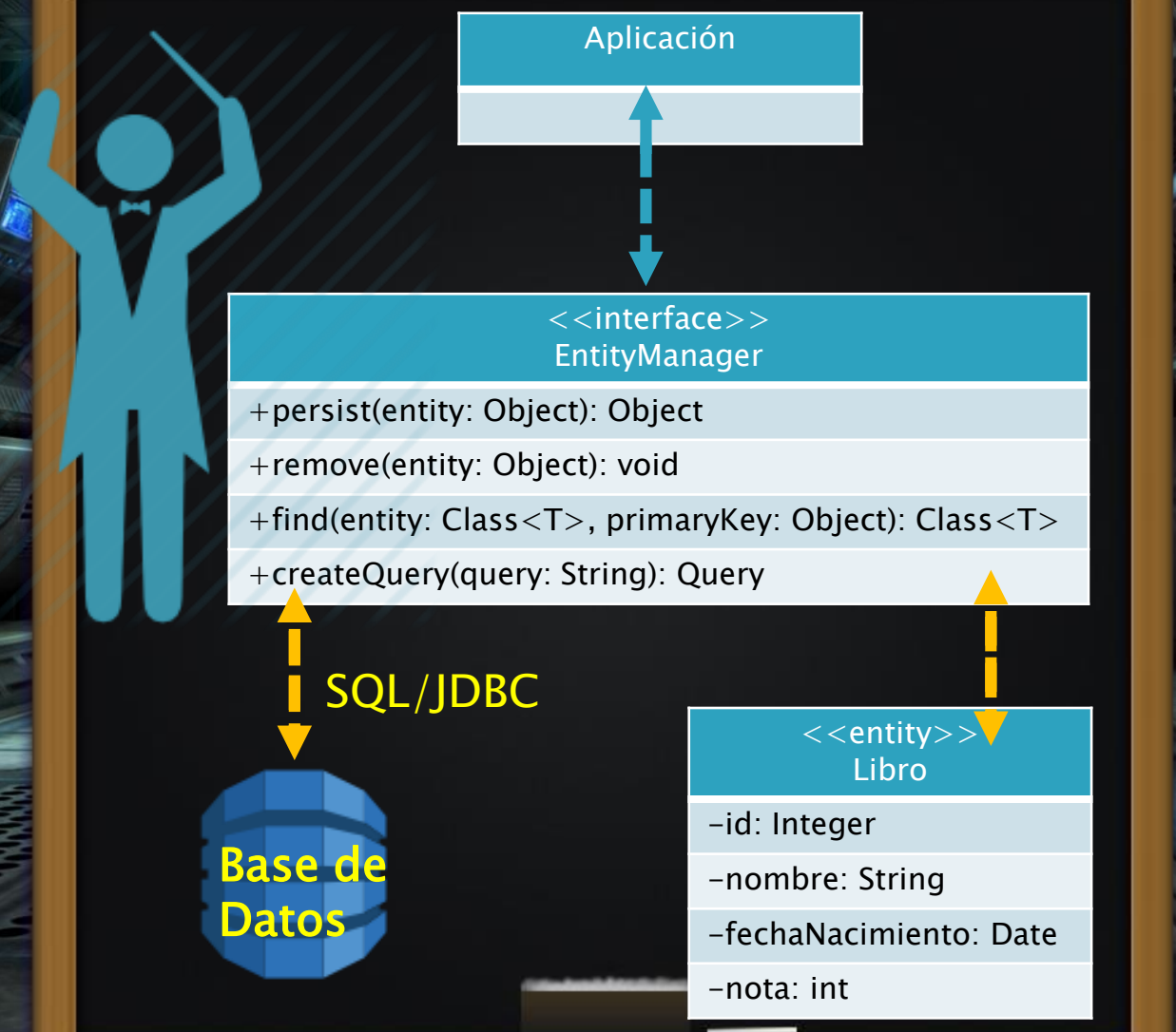
Dependencia Maven

```
<!-- Hibernate JPA-->
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-core</artifactId>
   <version>4.3.11.Final</version>
</dependency>

<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-entitymanager</artifactId>
   <version>4.3.11.Final</version>
</dependency>
```

JPA
Entity Manager

Clase Entity y el EntityManager



Es la interfaz en la que se apoya la API de persistencia y la que se encarga del mapeo entre una tabla relacional y su objeto de entidad Java (Entity)

Proporciona métodos para manejar la persistencia de un objeto de Entidad, permite agregar, eliminar, editar, consultar (JPQL) y manejar su ciclo de vida

En general se encarga de manejar o administrar el contexto de persistencia y sus entidades

Es el director de orquestas de las entidades JPA

JPA - Entity Manager

- Sus métodos más importantes son
  - persist(Object entity): Almacena un objeto entity en el contexto de persistencia y en la base de datos
  - merge(Object entity): Actualiza las modificaciones en una entidad devolviendo un objeto entity manejado por el contexto JPA
  - remove(Object entity): Elimina la entidad
  - find(Class<T> entity, Object primaryKey): Busca la entidad a través de su clave primaria

JPA - Entity Manager

- flush(): Sincroniza las entidades con la base de datos, hace el comit
- createQuery(String query): Crea una query utilizando el lenguaje JPQL o HQL
- createNativeQuery(): Crea una query utilizando el lenguaje SQL

Acceso al contexto de Persistencia

- Cualquier implementación DAO o Repositorio en Spring tendrán que acceder a un contexto de persistencia JPA
- Necesita acceder al EntityManager, por lo tanto es necesario Inyectar la referencia
- Podemos obtener una referencia al EntityManager a través de la anotación @PersistenceContext.

```
@PersistenceContext
private EntityManager entityManager;
```

Implementar DAO con JPA EntityManager

Implementación DAO usando JPA

```
@Repository("estudianteDao")
public class JpaEstudianteDao implements EstudianteDao {

    @PersistenceContext
    private EntityManager entityManager;

    @Transactional
    public void save(Estudiante estudiante) {
        entityManager.merge(estudiante);
    }

    @Transactional
    public void delete(Integer estudianteId) {
        Estudiante estudiante = entityManager.find(Estudiante.class, estudianteId);
        entityManager.remove(estudiante);
    }

    @Transactional(readOnly = true)
    public Estudiante findById(Integer estudianteId) {
        return entityManager.find(Estudiante.class, estudianteId);
    }

    @Transactional(readOnly = true)
    public List<Estudiante> findAll() {
        Query query = entityManager.createQuery("from Estudiante");
        return query.getResultList();
    }
}
```

Implementación DAO usando JPA

El DAO no tiene ninguna dependencia directa de Spring (salvo las transacciones), sin embargo se integra muy con Spring

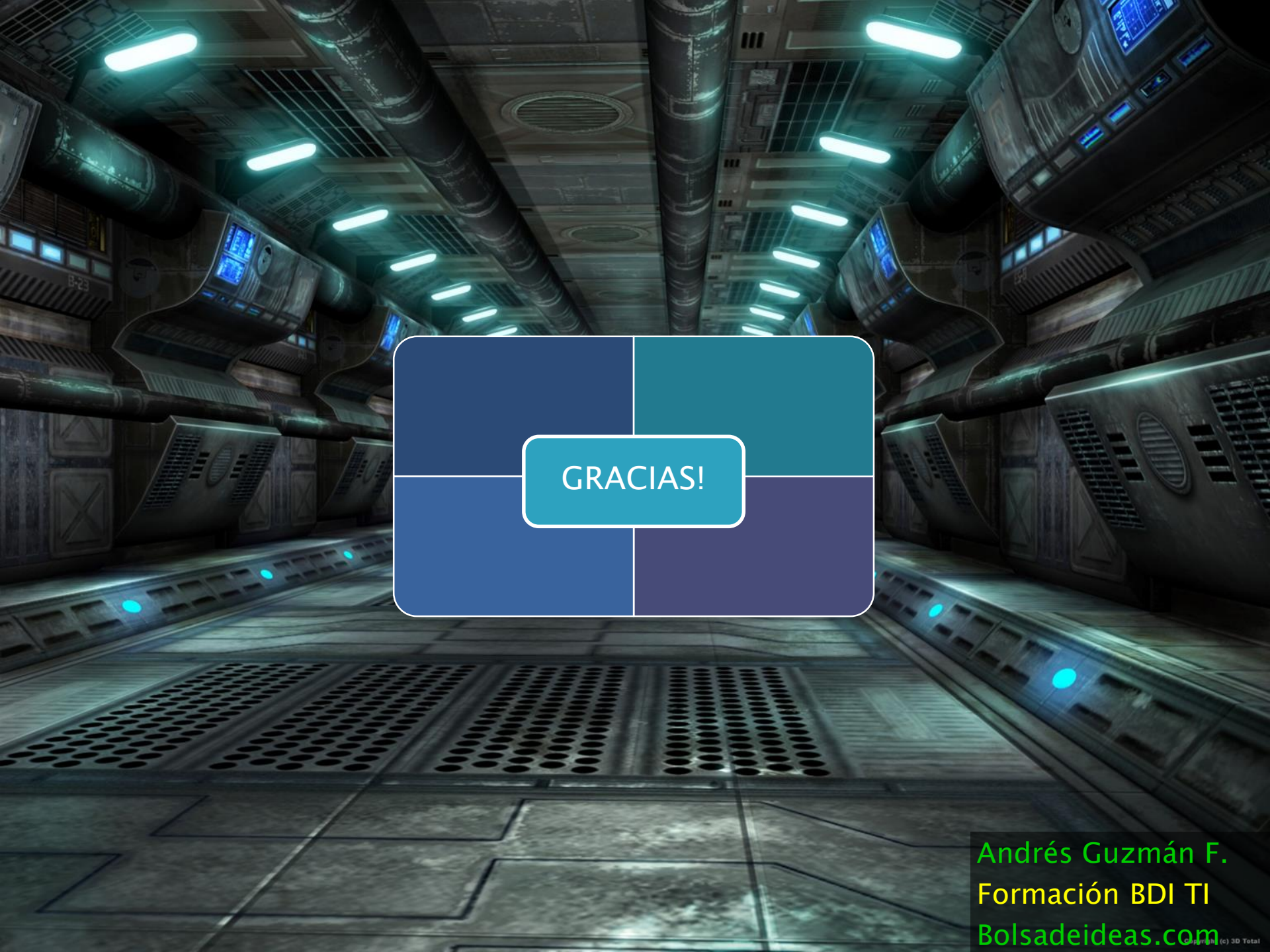
Además, el DAO utiliza la anotaciones @PersistenceContext (propia de Java EE) para requerir la inyección del EntityManager

Resumen de Dependencias Maven

```
<!-- Spring ORM -->
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-orm</artifactId>
   <version>4.2.0.RELEASE</version>
</dependency>

<!-- Hibernate -->
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-core</artifactId>
   <version>4.3.11.Final</version>
</dependency>
<!-- JDBC drivers -->
<dependency>
   <groupId>org.hsqldb</groupId>
   <artifactId>hsqldb</artifactId>
   <version>2.3.3</version>
</dependency>
<dependency>
   <groupId>mysql</groupId>
   <artifactId>mysql-connector-java</artifactId>
   <version>5.1.36</version>
</dependency>

<!-- DBCP2 -->
<dependency>
   <groupId>org.apache.commons</groupId>
   <artifactId>commons-dbcp2</artifactId>
   <version>2.1</version>
</dependency>
```



GRACIAS!

Andrés Guzmán F.
Formación BDI TI
Bolsadeideas.com