

## Concepto de ORM

<b>Concepto de ORM</b>	<b>1</b>
¿Qué aprenderás?	2
Introducción	2
Descripción de la tecnología	3
¿Por qué un ORM?	4
Mapeo Relacional	4
Hibernate	5
Mapeo de objetos	6



**¡Comencemos!**

## ¿Qué aprenderás?

- Conocer la tecnología ORM.
- Construir el mapeo relacional.

## Introducción

El proyecto anteriormente construido tiene una característica que explica por sí solo la razón por la cual es necesario trabajar con ORM, pero antes de entrar en detalles se hará una descripción de esta tecnología y sus alcances en el desarrollo de software.

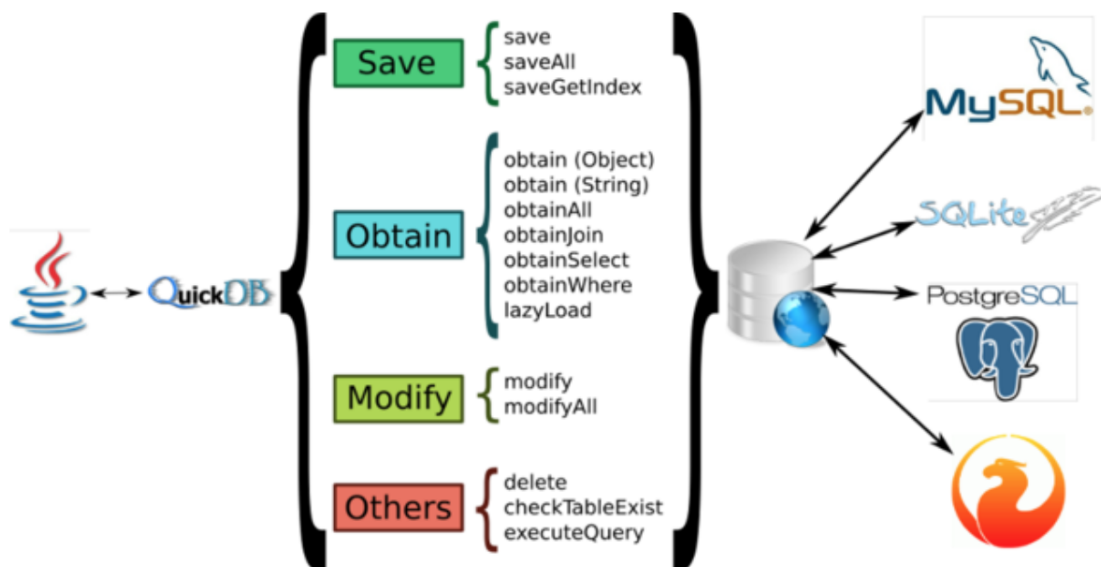


Imagen 1. Concepto de ORM.

Fuente: Desafío Latam.

## Descripción de la tecnología

En el principio de los tiempos, si se necesitaba acceder a una base de datos y trabajar con ella mediante un lenguaje orientado a objetos era necesario mezclar códigos y conceptos distintos entre sí, como por ejemplo en la clase DAO que se programó en el capítulo anterior.

Ya en estos niveles de programación hablamos de capas, y tenemos por un lado la capa de acceso a datos en la cual utilizamos ciertos objetos de las clases de conexión como la Connection, Query, etc. Para poder conectar y manipular la base de datos mediante consultas sql.

Estos elementos claramente son conceptos de la base de datos, por lo que poco y nada tienen que ver en el código orientado a objetos. Se está mezclando la responsabilidad en el archivo. Por otro lado, mediante código incrustado en string de java generamos las queries correspondientes. Otro problema es la representación de los datos, ya que difiere entre el motor de base de datos y el lenguaje de programación, generando inconsistencias en el tratamiento de los mismos.

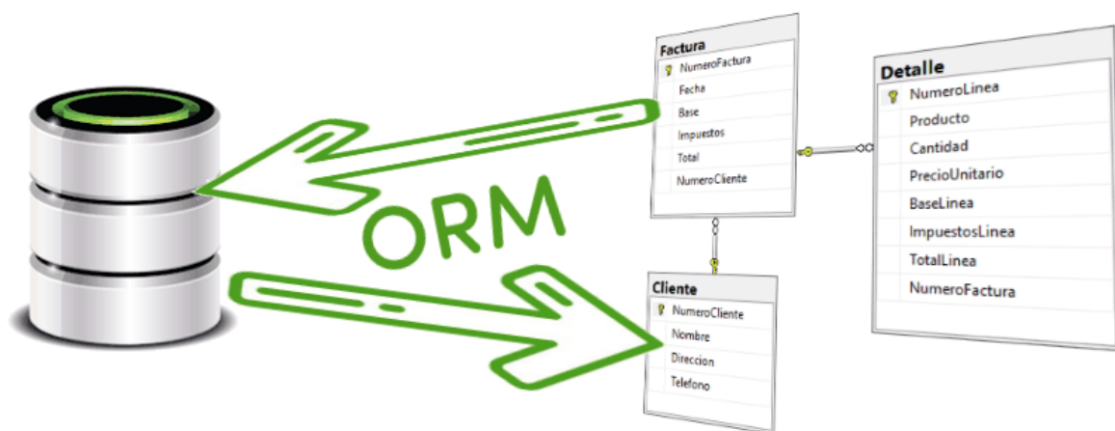


Imagen 2. ORM.  
Fuente: Desafío Latam.

## ¿Por qué un ORM?

En la actualidad es raro generar el manejo y administración de los datos desde un programa java con JDBC puro, ya que con esta técnica la simpleza del código pierde potencial y legibilidad al tener que generar las sentencias de sql directamente en código java, aportando brechas de seguridad que podrían ser aprovechadas por atacantes.

Para solventar estos problemas existen productos de software encargados de disminuir los riesgos y facilitar el diseño de la capa de datos mediante el mapeo de las mismas clases del sistema. Estos productos de software son conocidos como ORM (Object Relational Mapping).

## Mapeo Relacional

El mapeo relacional solamente se basa en generar “espejos” de las tablas del modelo de datos, creando clases con la misma estructura y atributos. Pensemos en un caso muy simple: Tenemos una clase Persona con los atributos nombre y apellido. Si se quiere mapear esta tabla en java, solamente se crea una clase de nombre Persona con dos atributos nombre y apellido.

Siguiendo el caso, ya que se tiene modelada en java la tabla usuarios, es posible generar métodos que manipulan la base de datos, pero sin ningún tipo de query, por ejemplo los métodos:

- buscarUsuarios();
- buscarUsuariosPorId(idUsuario);
- eliminarUsuario(idUsuario);

Por citar algunos. Como se ve en ningún momento la aplicación trata con código sql, si no que es el ORM el encargado de generar las sentencias, siendo transparentes para el sistema. Ese es el poder de un ORM y existen de distintos tipos y fabricantes, siendo el más conocido Hibernate.

## Hibernate

Hibernate es una biblioteca de mapeo relacional de objetos de código abierto para java, diseñada para mapear objetos a un sistema de base de datos relacional y así implementar los conceptos de programación orientada a objetos en una base de datos.

Tiene varias diferencias con el estándar JDBC entre los cuales destacan:

- A diferencia de JDBC, hibernate se conecta con la base de datos y usa HQL (Hibernate Query Language) para ejecutar todas las consultas, para luego asignar los resultados a los objetos java.
- Hibernate se configura mediante xml facilitando su gestión, ya que permite configurar los objetos que serán administrados por el framework.
- Mejor manejo de la conexión, ya que trabaja mediante la sesión de la aplicación.
- Nos olvidamos de controlar la instancia de conexión, ya que cuenta con un *session factory* que administra la conexión como si de un singleton se tratara.

## Mapeo de objetos

Al utilizar JDBC, es necesario escribir código para asignar la representación de los datos del modelo a objetos de un modelo relacional y su esquema correspondiente. Hibernate en cambio asigna directamente las clases de java a las tablas de la base de datos utilizando xml o anotaciones. En el siguiente ejemplo se intentará explicar la diferencia entre JDBC e Hibernate.

### Implementación con JDBC

```
while(resultado.next()) {  
    CursoDTO cursoDto = new CursoDTO();  
  
    cursoDto.setIdCurso(resultado.getInt("id_curso"));  
  
    cursoDto.setDescripcion(resultado.getString("descripcion"));  
  
    cursoDto.setPrecio(resultado.getDouble("precio"));  
    listaDeCursos.add(cursoDto);  
}
```

### Implementación con Hibernate

```
@Entity  
@Table(name = "curso")  
public class CursoDTO {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private int idCurso;  
  
    private String descripcion;  
  
    private double precio;  
  
    public int getIdCurso() {  
        return idCurso;  
    }  
    public void setIdCurso(int idCurso) {  
        this.idCurso = idCurso;  
    }  
    public String getDescripcion() {  
        return descripcion;  
    }  
}
```

```
public void setDescripcion(String descripcion) {  
    this.descripcion = descripcion;  
}  
public double getPrecio() {  
    return precio;  
}  
public void setPrecio(double precio) {  
    this.precio = precio;  
}  
}
```

En el fragmento de código con Hibernate se pueden apreciar una serie de anotaciones que trabajan sobre la misma clase DTO que representa a la tabla curso, y no es necesario establecer todas las propiedades de un objeto al obtener los datos.

Vamos a ver como funciona un ORM con un ejemplo utilizando Hibernate con anotaciones. Seguir los pasos con cuidado, ya que al momento de utilizar frameworks uno puede perder mucho tiempo en configuraciones. En este ejercicio se utilizara maven, ya que debemos importar Hibernate y varias otras librerías y siguiendo la forma tradicional, es fácil volverse locos con tantos .jar que hay que referenciar.

Primero creamos un proyecto java maven, sin ningún arquetipo.

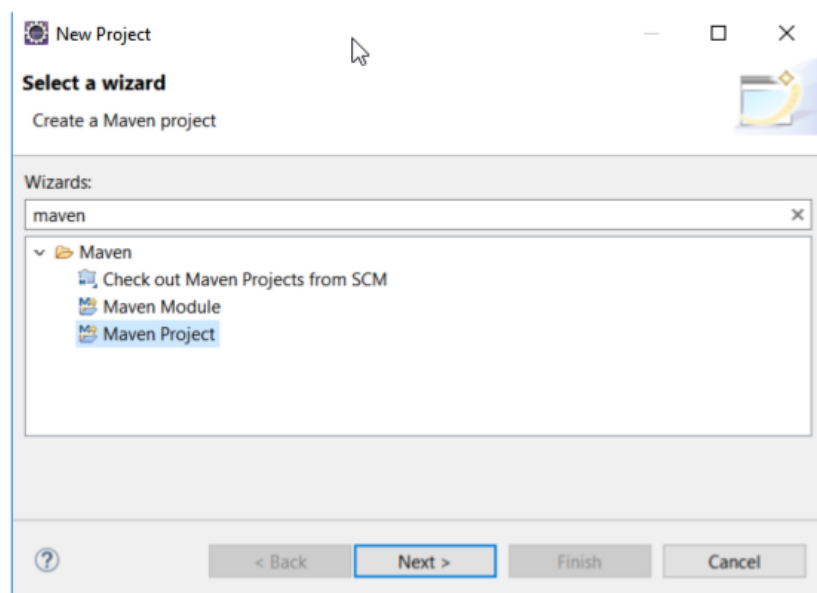


Imagen 3. File-New Project-Maven project.

Fuente: Desafío Latam.

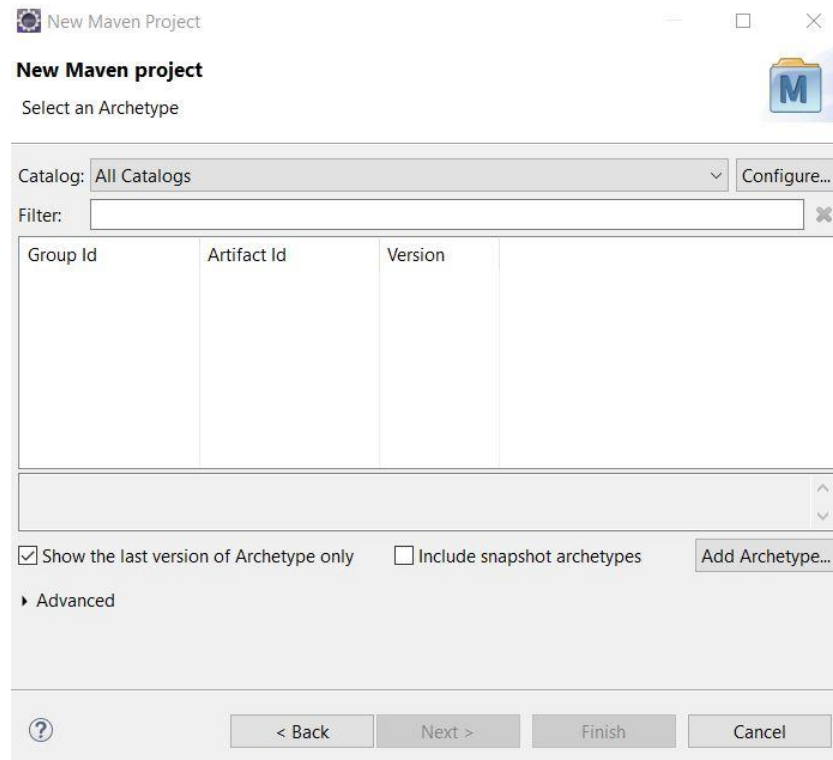


Imagen 4. Seleccionamos el "Archetype".  
Fuente: Desafío Latam.



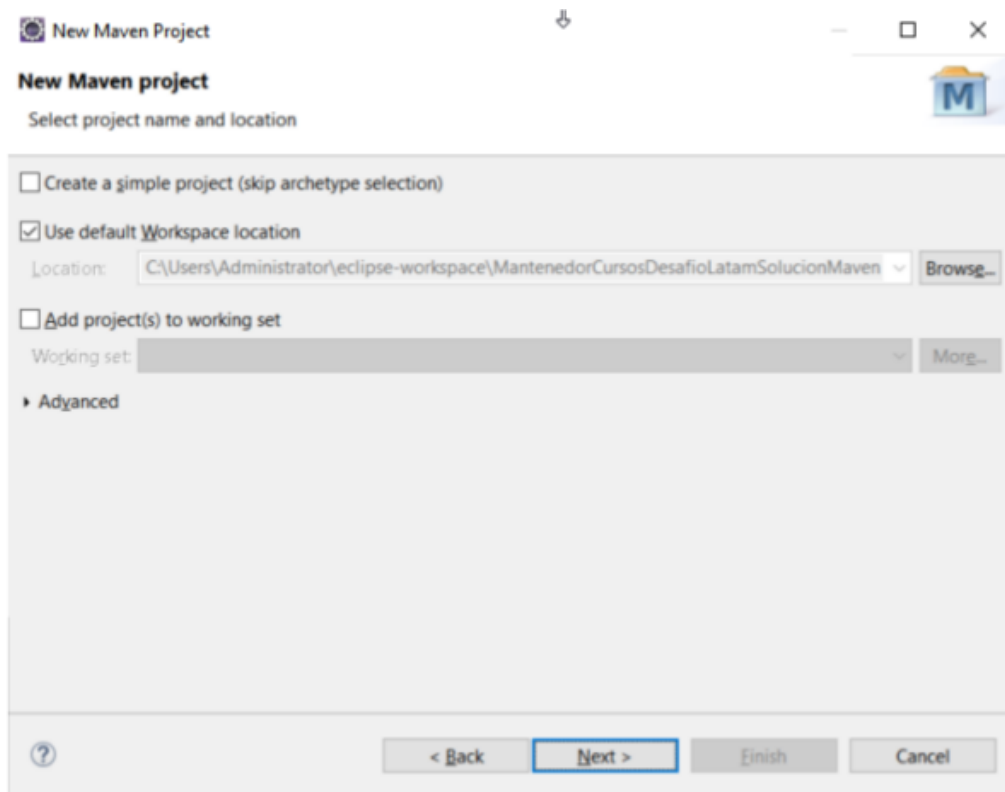
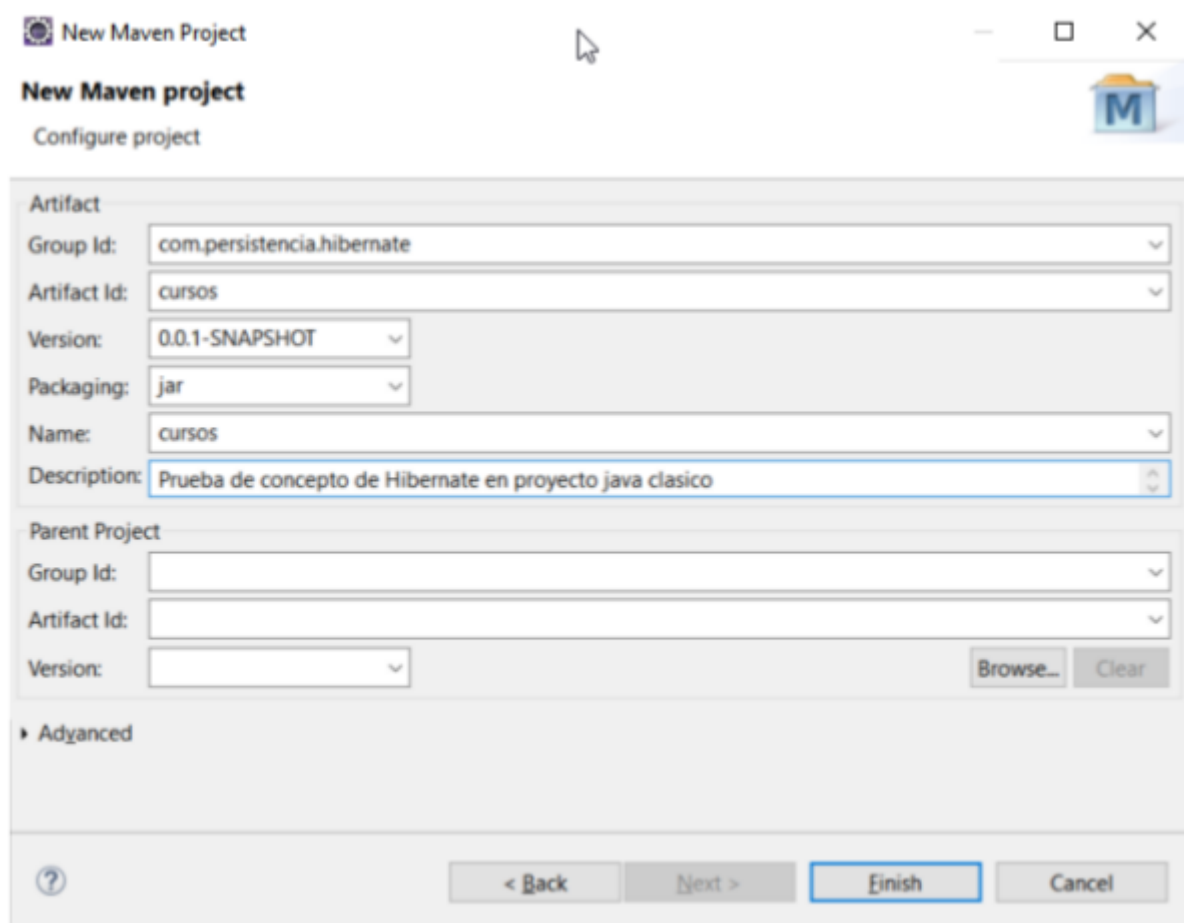


Imagen 5. Sólo marcar las opciones señaladas.  
Fuente: Desafío Latam.



New Maven Project

**New Maven project**  
Configure project

Artifact

Group Id:

Artifact Id:

Version:

Packaging:

Name:

Description:

Parent Project

Group Id:

Artifact Id:

Version:

► Advanced

Imagen 6. Los campos 'Name' y 'Description' son opcionales.  
Fuente: Desafío Latam.

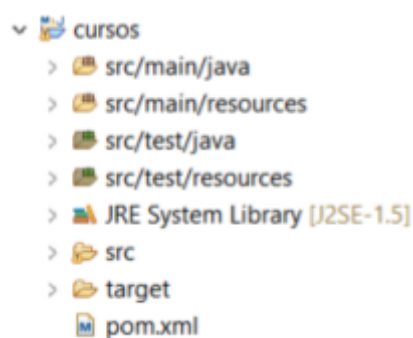


Imagen 7. Generación del proyecto maven.  
Fuente: Desafío Latam.

Ya está el proyecto creado. Ahora es tiempo de usar el poder de maven para poder usar Hibernate en el proyecto. Abrir el archivo *pom.xml* y añadir el siguiente código:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.persistencia.hibernate</groupId>
  <artifactId>cursos</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>cursos</name>
  <description>Prueba de concepto de Hibernate en proyecto java
clasico</description>
  <build>
    <sourceDirectory>src</sourceDirectory>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.0</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
        <version>3.2.3</version>
        <configuration>

<warSourceDirectory>WebContent</warSourceDirectory>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
      <version>3.1.0</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
```

```
        <artifactId>hibernate-annotations</artifactId>
        <version>3.4.0.GA</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>javassist</groupId>
        <artifactId>javassist</artifactId>
        <version>3.4.GA</version>
        <scope>compile</scope>
    </dependency>

</dependencies>
</project>
```

El *pom.xml* se encarga de disponibilizar:

- maven-compiler-plugin 3.8.0
- maven-war-plugin 3.2.3
- javax.servlet-api 3.1.0
- hibernate-annotations 3.4.0.GA
- javassist 3.4.ga

Antes de ejecutar maven también debemos añadir el driver de oracle, siguiendo esta ruta:

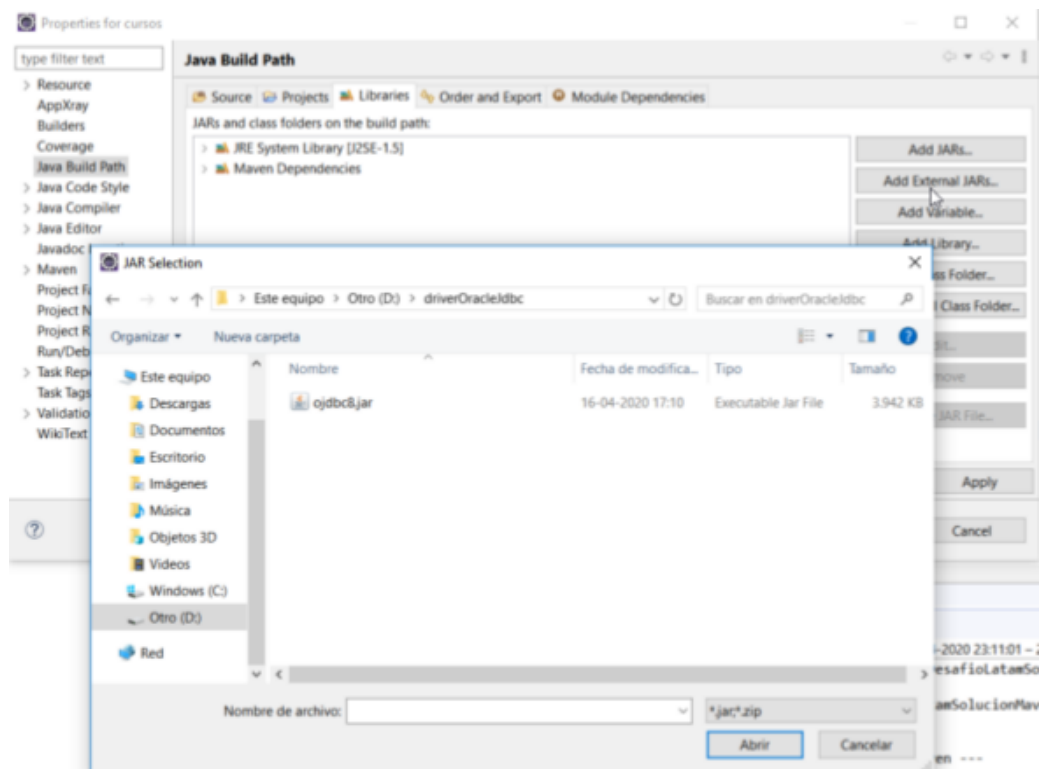


Imagen 8. Añadir librería ojdbc8.jar.

Fuente: Desafío Latam.

Ahora ejecutamos el comando *clean install*:

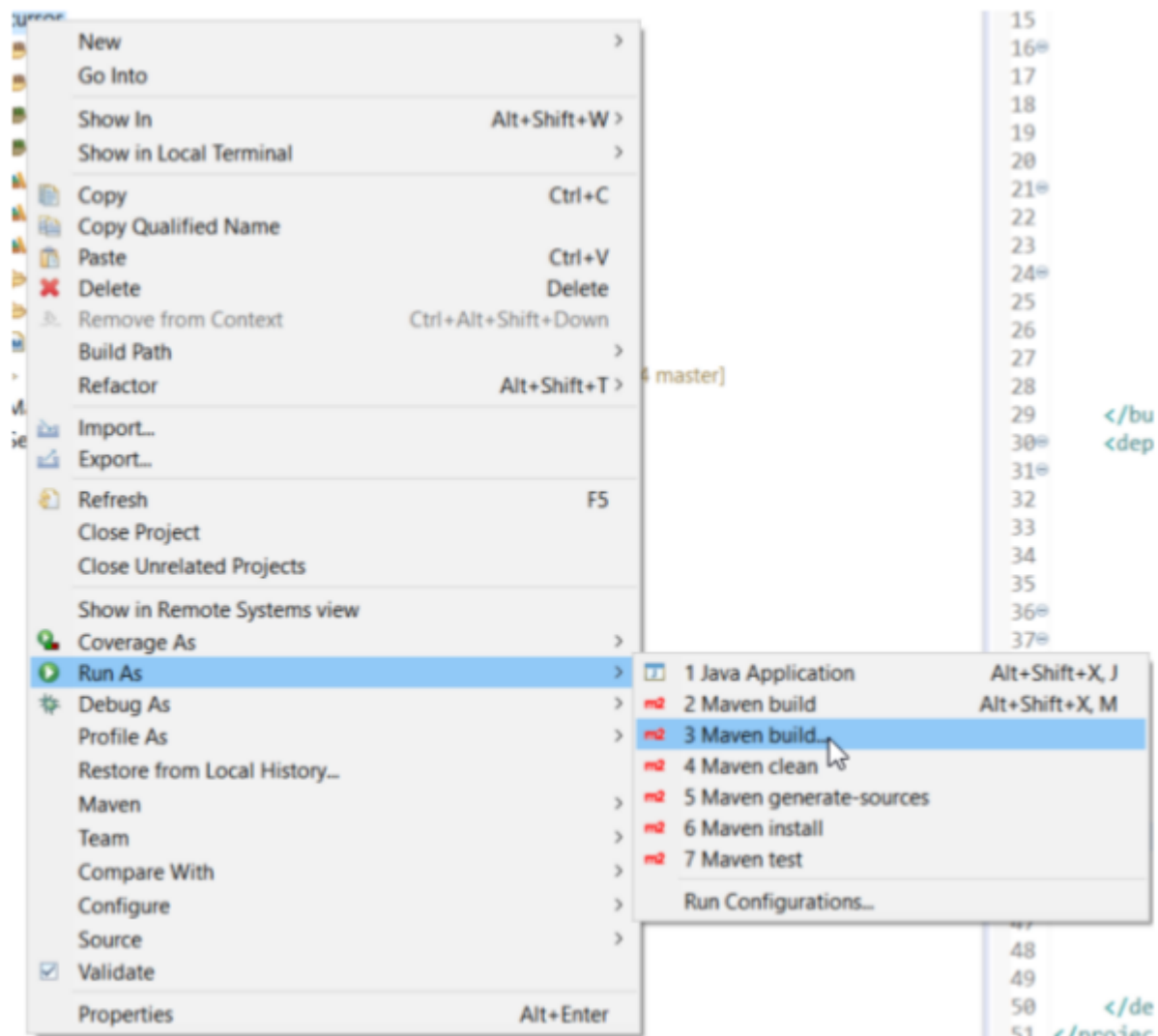


Imagen 9. Ejecutamos el maven.

Fuente: Desafío Latam.

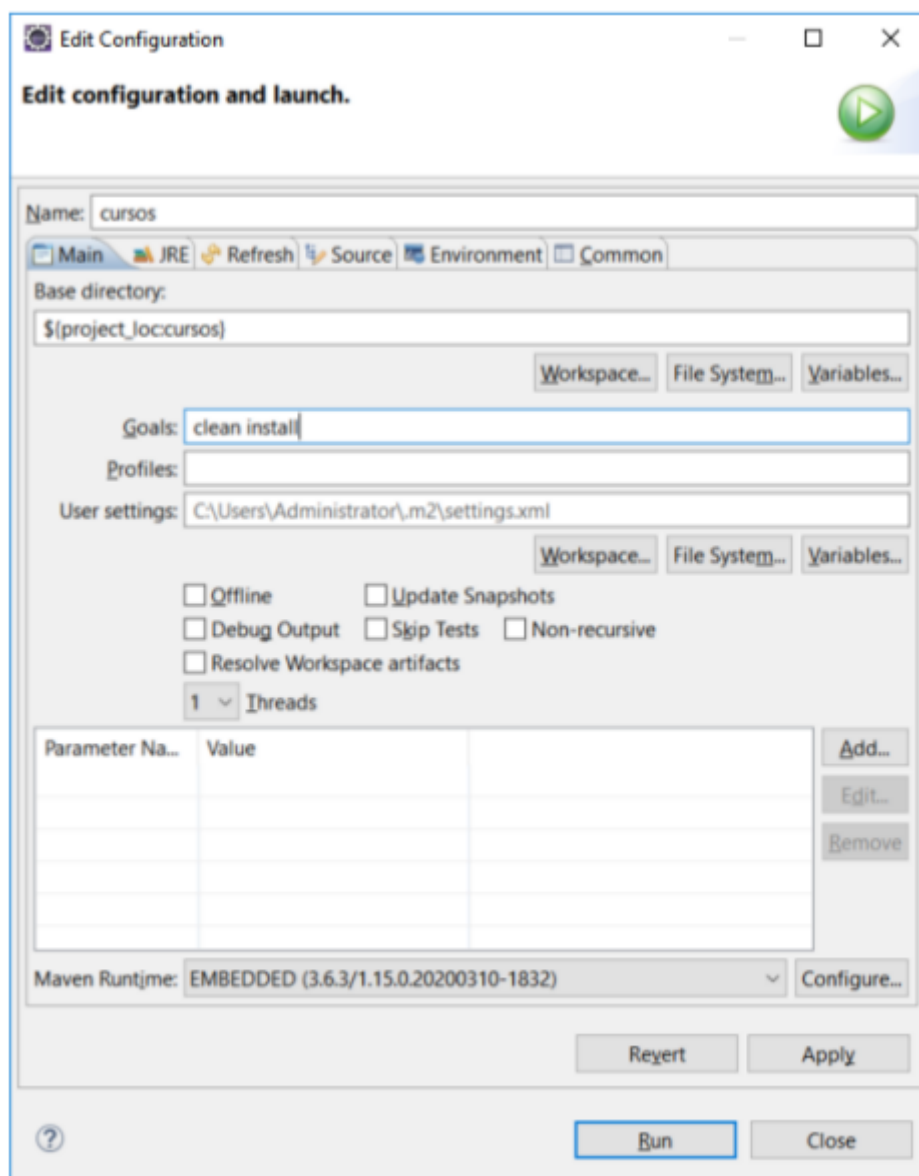


Imagen 10. Clean install.

Fuente: Desafío Latam.

La consola de eclipse debería correr con éxito. Con esto tenemos el proyecto configurado y listo para empezar con Hibernate. Se utilizará la misma base de datos que se ha estado trabajando en los ejemplos y desafíos. Vamos a hacer una inserción de un curso nuevo a la tabla 'curso' utilizando el ORM.

Siguiendo la lógica del proyecto *MantenedorCursos*, nosotros ya sabemos que existe en tal proyecto varias clases de tipo DTO las cuales son un fiel reflejo de las tablas de la base de datos y en este caso también replicaremos esa estructura pero solo de la clase CursoDTO. En este proyecto la clase solo se llamará Curso. Creamos un package de nombre entidades y creamos la clase.

```
package entidades;

import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.SequenceGenerator;
import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name = "curso")
public class Curso {
    private int idCurso;
    private String descripcion;
    private double precio;

    public Curso() {
    }

    @Id
    @Column(name = "id_curso")
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
generator="cursoid")
    @SequenceGenerator(name="cursoid", sequenceName="cursoid",
allocationSize=1)
    public int getIdCurso() {
        return idCurso;
    }

    public void setIdCurso(int idCurso) {
        this.idCurso = idCurso;
    }

    @Basic
    @Column(name = "descripcion")
    public String getDescripcion() {
        return descripcion;
    }

    public void setDescripcion(String descripcion) {
        this.descripcion = descripcion;
    }

    @Basic
    @Column(name = "precio")
    public double getPrecio() {
```



```
        return precio;
    }
    public void setPrecio(double precio) {
        this.precio = precio;
    }
}
```

A simple vista es solo un POJO con los atributos idCurso, descripción y precio, pero además existe un nuevo constructor vacío. Este constructor es necesario para que Hibernate pueda acceder y trabajar con el mapeo de objetos. También se añaden las anotaciones:

- **@Entity**: Convierte la clase que la acompaña en un bean de entidad, para que Hibernate pueda reconocerla y así trabajar persistentemente.
- **@Table**: Anotación que hace referencia a la tabla de base de datos que la clase representa. Si se omite, Hibernate asumirá que la tabla tiene el nombre de la clase.
- **@Id**: Anotación que identifica a la primary key de la tabla, en este caso IdCurso. También lo acompaña la anotación **@Column(name="")** que indica cual es la columna que referencia. Las anotaciones **@GeneratedValue** y **SequenceGenerator** trabajan con una secuencia que maneja Hibernate para aumentar en 1 la primary key en cada inserción. En la base de datos es necesario crear la secuencia para que pueda trabajar.
- **@Basic**: En cada 'getter' se puede añadir esta anotación para indicar que son atributos normales de la base de datos. También puedes indicar cual es el nombre de la columna y al igual que con **@Table**, si no se especifica Hibernate asume que el atributo se llama igual que la variable.

Ahora que tenemos la clase que se mapeara, se procede a configurar Hibernate. Cuando se utilizan anotaciones hay que generar un archivo *.xml* que maneja la conexión a la base de datos y las clases que el framework tendrá que administrar. El archivo se llama *hibernate.cfg.xml*.

```
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- Database connection settings -->
        <property
```

```
name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
<
    <property
name="connection.url">jdbc:oracle:thin:@//localhost:1521/xe</property>
    <property name="connection.username">Empresa_DB</property>
    <property name="connection.password">empresa</property>
    <!-- JDBC connection pool (use the built-in) -->
    <property name="connection.pool_size">1</property>
    <!-- SQL dialect -->
    <property
name="dialect">org.hibernate.dialect.Oracle8iDialect</property>
    <!-- Enable Hibernate's automatic session context management
-->
    <property
name="current_session_context_class">thread</property>
    <!-- Disable the second-level cache -->
    <property
name="cache.provider_class">org.hibernate.cache.NoCacheProvider</propert
y>
    <!-- Echo all executed SQL to stdout -->
    <property name="show_sql">true</property>
    <mapping class="entidades.Curso"/>
</session-factory>
</hibernate-configuration>
```

En el archivo se pueden identificar el driver de conexión, la url, el user y pass del usuario de base de datos y en la línea <mapping> se declara la o las clases que deben ser mapeadas por Hibernate, en este caso solo mapeamos la clase Curso.

A continuación creamos una clase de nombre *HibernateExe.java* que se encargará de generar una sesión basada en el archivo de configuración *hibernate.cfg.xml*.

```
package entidades;

import java.io.File;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
    private static final SessionFactory sessionFactory;
```

```
        static {
            try {
                // Create the SessionFactory from hibernate.cfg.xml
                sessionFactory = new
AnnotationConfiguration().configure().buildSessionFactory();
            } catch (Throwable ex) {
                // Make sure you log the exception, as it might be
swallowed
                System.err.println("Initial SessionFactory creation
failed." + ex);
                throw new ExceptionInInitializerError(ex);
            }
        }

        public static SessionFactory getSessionFactory() {
            return sessionFactory;
        }
    }
}
```

Y por último, la clase *Main* que creará la sesión y por fin hará la inserción a la base de datos.

```
package entidades;

import org.hibernate.Session;

public class TestDaoHbm {

    public static void main(String[] args) {
        Session session =
HibernateUtil.getSessionFactory().getCurrentSession();
        session.beginTransaction();
        Curso curso = new Curso();
        curso.setDescripcion("Spring");
        curso.setPrecio(580000);
        session.save(curso);
        session.getTransaction().commit();
        System.out.println("curso insertado "+
curso.getDescripcion());
    }
}
```

La estructura del proyecto debe verse así, ojo con la ubicación de los archivos.

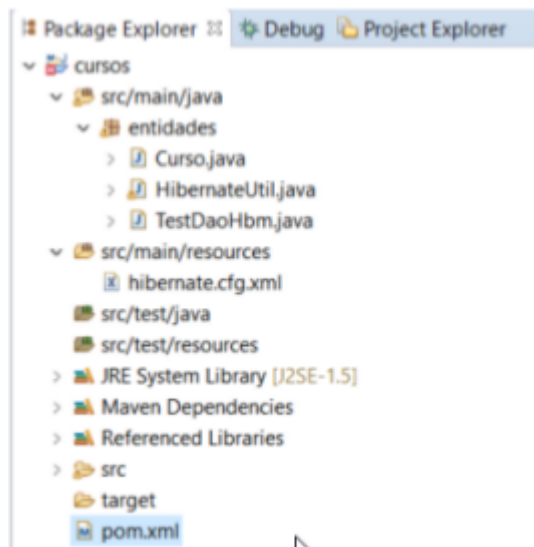


Imagen 11. Estructura de carpetas.

Fuente: Desafío Latam.

Ahora podemos ejecutar la clase main, y ver como hibernate genera la inserción sin necesidad de nosotros escribir un insert con *sql* directamente. Ver las clases DAO del proyecto anterior y comparar la limpieza de código en la implementación.

```

<terminated> TestDaoHbm (1) [Java Application] C:\Program Files\Java\jdk1.8.0_211\bin\javaw.exe (20-04-2020 00:36:36 - 00:36:38)
log4j:WARN No appenders could be found for logger (org.hibernate.cfg.annotations.Version).
log4j:WARN Please initialize the log4j system properly.
Hibernate: select cursoid.nextval from dual
Hibernate: insert into curso (descripcion, precio, id_curso) values (?, ?, ?)
curso insertado Cocina
    
```

ID_CURSO	DESCRIPCION	PRECIO
1 14	Cocina	580000

Imagen 12. Funcionamiento básico de Hibernate.

Fuente: Desafío Latam.

Esta implementación demuestra el funcionamiento básico de Hibernate. Si bien la configuración puede ser algo confusa y solo es una pincelada de todo el poder del framework, es posible dejar de usar JDBC en un proyecto. En módulos posteriores se hará uso de Hibernate.