

Bootcamp Java Developer

Fase 3 - Java Architect
Módulo 30



Hibernate

Hibernate

Es un *framework* – construido en Java - que se encarga de manejar la persistencia de datos.

Es la implementación de un ORM.

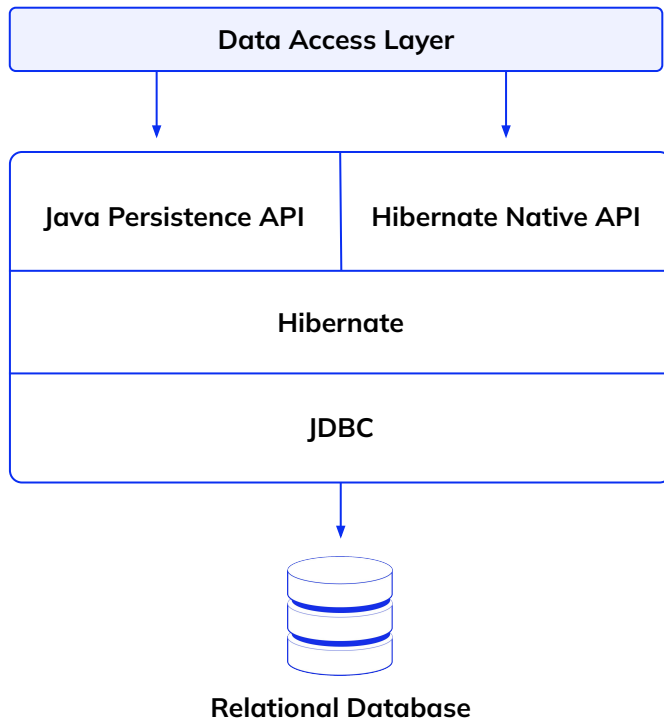
Es un proyecto de código abierto no comercial (inicialmente), bajo los aspectos de la licencia pública GNU. En el año 2003 se une con JBoss.org y consigue su lado comercial: venta de soporte y capacitación.

Los fuentes/binarios están disponibles en [Hibernate.org](https://hibernate.org)

Tiene como objetivo ser una solución completa a la problemática de persistencia de datos con **tecnología Java**, y hace posible que el desarrollador se concentre fundamentalmente en los aspectos de negocio. Aumenta fuertemente la productividad y facilita la administración de la capa de acceso a datos.

Uno de los aspectos importantes de Hibernate es que **está construido con clases Java básicas**, es decir que **se puede agregar el *framework* a un proyecto propio**. No se requiere ningún tipo de *container* (por ejemplo, un *application server*).

Hibernate



La necesidad de una DAL (Data Access Layer)

Arquitectura multicapa (N-Tier Architecture)

Es una forma de organizar un sistema. Cada capa está compuesta, generalmente, por un **conjunto de clases que cooperan con un objetivo en común. Cada capa le brinda servicios a la capa superior directa**, no se realizan llamados entre capas “*que no se ven*”.



Una de las arquitecturas más utilizadas es la **arquitectura 3 capas**:

- **PL** (*Presentation Layer*).
- **BL** (*Business Layer* o *Business Logic*).
- **DAL** (*Data Access Layer*).

Se la denomina *3TA* o *3-Tier Architecture*, y provee una separación lógica (no necesariamente física) entre las 3 capas.

- **PL o Capa de Presentación:** se encarga de administrar la lógica correspondiente para mostrar la información en pantalla. Generalmente, viene acompañada del uso del patrón de diseño MVC.
- **BL o Capa de Negocios:** se encarga de administrar la lógica de negocios, aquí es donde reside la implementación de las reglas de negocio (*Business Rules*).
- **DAL o Capa de Acceso a Datos:** se encarga de manejar cualquier comunicación con la base de datos, ya sea para obtener información como para escribir.

Por ejemplo

Basándonos en tecnología Java, la PL podría estar construida con JSP y Struts, la BL con EJB de tipo session, y la DAL con Hibernate.



¿Qué es POJO?

Es un **Plain Old Java Object**, aunque también se lo denomina *Plain Ordinary Java Object*. Se trata de **una clase que se encarga de manejar la persistencia**. Persigue las mismas características que los *Java Beans*, es decir:

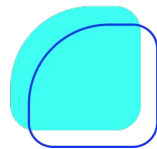
- Por lo general, es **serializable**, e implementa la interfaz (vacía) `Serializable`.
- Posee **atributos privados y métodos públicos de acceso**.
- Posee un **constructor vacío**, lo cual es obligatorio para trabajar con Hibernate.

Ejemplo

```
[code]
public class Usuario implements Serializable {
    private String nombre;
    private String clave;

    public Usuario(){
    }

    // Setters y getters
        . . .
        . . .
}
[/code]
```



Ventajas de una arquitectura Multi-Capa

- Provee una **separación lógica** entre las grandes tareas del sistema.
- Permite que cada capa se concentre en una **única tarea**.
- Aumenta la **organización**.
- Aumenta la **escalabilidad**.
- Facilita el **mantenimiento**.



Introducción a un proyecto con Hibernate

Configuración

Para configurar un proyecto con Maven, es necesario agregarlo en el **pom.xml**:



```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.6.RELEASE</version>
  <relativePath /> <!-- lookup parent from repository -->
</parent>

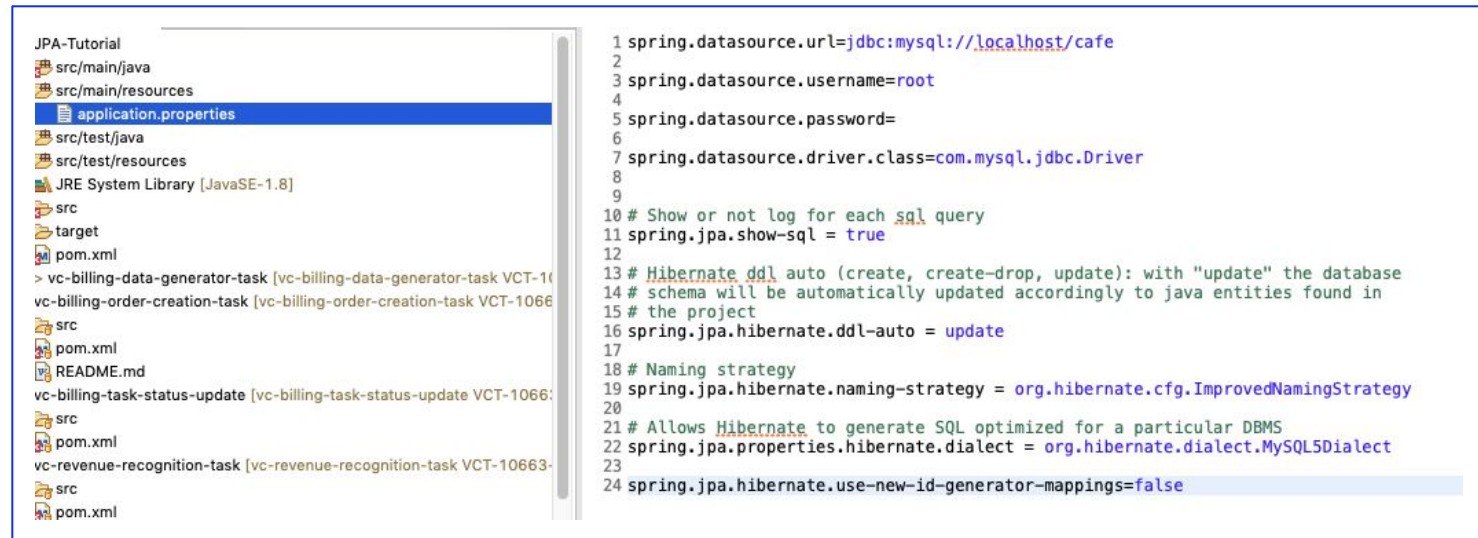
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  <java.version>1.8</java.version>
</properties>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>

  <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <!--version>5.1.47</version-->
  </dependency>
```

Luego es necesario crear un archivo llamado ***application.properties***, que es el que utiliza Hibernate para configurar su funcionamiento. Debe agregarse en la carpeta **src/main/resources**.



The screenshot shows an IDE with a file explorer on the left and a code editor on the right. The file explorer displays the project structure for 'JPA-Tutorial', with 'src/main/resources/application.properties' selected. The code editor shows the content of this file, which is a Spring Boot configuration for a MySQL database.

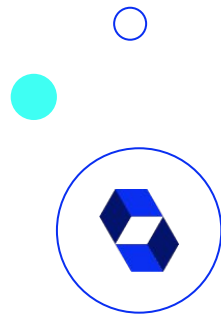
```
1 spring.datasource.url=jdbc:mysql://localhost/cafe
2
3 spring.datasource.username=root
4
5 spring.datasource.password=
6
7 spring.datasource.driver.class=com.mysql.jdbc.Driver
8
9
10 # Show or not log for each sql query
11 spring.jpa.show-sql = true
12
13 # Hibernate ddl auto (create, create-drop, update): with "update" the database
14 # schema will be automatically updated accordingly to java entities found in
15 # the project
16 spring.jpa.hibernate.ddl-auto = update
17
18 # Naming strategy
19 spring.jpa.hibernate.naming-strategy = org.hibernate.cfg.ImprovedNamingStrategy
20
21 # Allows Hibernate to generate SQL optimized for a particular DBMS
22 spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect
23
24 spring.jpa.hibernate.use-new-id-generator-mappings=false
```

Propiedades de Hibernate

- **`spring.datasource.driver.class`** representa el *driver* a utilizar. Corresponde al nombre de la clase que implementa el *driver* JDBC, incluido en el .jar correspondiente. Dicho .jar, como ha sido especificado anteriormente, deberá formar parte del CLASSPATH de la aplicación.
- **`spring.datasource.url`** representa la URL de conexión a utilizar. Especifica el *host*, el puerto y la base de datos a utilizar. La URL de conexión depende directamente del DBMS.
- **`spring.datasource.username`** representa el usuario en la conexión a utilizar.
- **`spring.datasource.password`** representa la contraseña en la conexión a utilizar.



- **`spring.jpa.properties.hibernate.dialect`** representa el dialecto (el lenguaje) a utilizar. Es necesario determinarlo ya que la implementación de SQL es distinta en cada uno de los DBMS propietarios. El dialecto depende directamente del DBMS.
- **`spring.jpa.show-sql`** especifica si mostramos como hibernate genera cada *query*.
- **`spring.jpa.hibernate.naming-strategy`** permite definir una estrategia para los nombres de las tablas.
- **`spring.jpa.hibernate.use-new-id-generator-mappings`** permite definir una estrategia para los nombres de las claves primarias.



- **spring.jpa.hibernate.ddl-auto** especifica si queremos que hibernate modifique la base de datos, basándose en las entidades de Java. Esto es muy útil en desarrollo pero no recomendable para Producción.

Las **opciones** son:

validate	Valida el schema, no cambia la base de datos.
update	Cambia la base de datos.
create	Crea el schema, destruyendo datos anteriores.
create-drop	Igual que el anterior pero además dropea el schema cuando se cierra la aplicación.

Hibernate/JPA Annotations

¿Qué son las Hibernate/JPA Annotations?

Existen dos formas para realizar la transformación de información de un POJO a la base de datos y viceversa. **Una es mediante archivos XML y otra es con Annotations.** Cada una tiene sus pros y sus contras, sin embargo las Annotations son la forma más nueva de realizar el mapeo de POJOs a la base de datos, en este curso utilizaremos la metodología de Annotations para la realización de todos los ejercicios.

Cuando se utiliza **Hibernate Annotations** toda la metadata está embebida en la misma clase Java que contiene al POJO, junto con el código Java, esto ayuda al desarrollador a relacionar la estructura de la tabla y del POJO al mismo tiempo que se escribe el código.



@Entity

A partir de esta *annotation* indicamos a Hibernate que nuestra clase es persistente.

Se recomienda el uso de la *annotation* Entity de JPA, la de Hibernate suele traer dificultades.

Una vez declarada persistente nuestra clase, todos sus atributos lo serán también por *default*.

@Table(name = "ITEM")

A través de esta *annotation* especificamos un nombre de tabla en caso de diferir al de nuestra clase. De no indicarlo, Hibernate tomará como nombre de la tabla asociada, el mismo que el de la clase. Esta es una convención, y constituye un ejemplo sencillo del concepto "*convention over configuration*".



@Id

A través de esta *annotation* indicamos que la *property* mapeada será la PK de nuestra tabla. En este caso se trata de una clave simple.

@GeneratedValue

Especifica la estrategia de generación de *IDs*. Hay varias opciones para generadores. En nuestro caso, al no acompañarla de parámetros adicionales, su valor será *AUTO*, que decide una estrategia de generación de *IDs* conveniente dependiendo del motor subyacente (ej. *Identity*).

@Column(name = "ITEM_ID")

En este caso la *annotation* es opcional, si pretendemos que la columna de la tabla correspondiente se llame igual que la *property*. Aquí valen las mismas consideraciones que en el caso de *@Table*.

@Repository

Es una anotación de Spring que especifica que es una clase que trabaja con entidades y es levantada por el *container* de Spring como un *bean*.

**¡Sigamos
trabajando!**