

Hibernate

... DOCUMENTO EN DESARROLLO ...

Introducción

Hibernate es un poderoso **framework de mapeo objeto-relacional (ORM)** para Java que simplifica significativamente el proceso de interacción con bases de datos relacionales. Desarrollado inicialmente por Gavin King en 2001 y actualmente mantenido por Red Hat, Hibernate se ha convertido en una herramienta fundamental para desarrolladores Java que buscan gestionar la persistencia de datos de manera eficiente y transparente.

En términos simples, Hibernate permite a los desarrolladores trabajar con objetos Java en lugar de consultas SQL directas. Esto se logra mediante el mapeo de objetos Java a tablas de bases de datos y viceversa, facilitando así la manipulación y persistencia de datos en la aplicación. Hibernate se integra estrechamente con el entorno de desarrollo Java y es compatible con una amplia gama de bases de datos relacionales, lo que lo convierte en una solución flexible y escalable para aplicaciones empresariales.

Entre sus características principales se incluyen:

- **Mapeo Objeto-Relacional (ORM):** Hibernate simplifica la representación de objetos Java como entidades persistentes en bases de datos relacionales, gestionando automáticamente la correspondencia entre las estructuras de datos y los objetos en el código.
- **Transacciones y gestión de la sesión:** proporciona un mecanismo robusto para administrar transacciones de bases de datos y sesiones de trabajo, asegurando la integridad y consistencia de los datos.
- **Consultas orientadas a objetos:** permite a los desarrolladores realizar consultas utilizando el lenguaje de consultas orientado a objetos de Hibernate (HQL), que es más intuitivo y menos propenso a errores que el SQL estándar.
- **Caché de segundo nivel:** Hibernate ofrece soporte para cache de segundo nivel, lo que mejora el rendimiento al reducir la cantidad de consultas repetitivas a la base de datos.
- **Soporte para herencia y asociaciones:** permite modelar relaciones complejas entre entidades y soporta herencia en la capa de persistencia, facilitando el diseño de bases de datos relacionales más complejas.

En resumen, Hibernate simplifica el desarrollo de aplicaciones Java al proporcionar una capa de abstracción entre la lógica de negocio y la capa de persistencia de datos, permitiendo a los desarrolladores centrarse en la funcionalidad de la aplicación sin tener que preocuparse por los detalles de la manipulación de la base de datos. Esto lo convierte en una herramienta indispensable en el arsenal de cualquier desarrollador Java que trabaje con bases de datos relacionales.

Sección generada por ChatGPT

Configuration and bootstrap

Including Hibernate in your project build

Para utilizar Hibernate fuera del entorno de un contenedor como WildFly o [Quarkus](#), se necesita:

- incluir el propio Hibernate ORM, junto con el controlador JDBC apropiado, como dependencias del su proyecto
- configurar Hibernate con información sobre la base de datos, especificando propiedades de configuración.

Añadir la dependencia de Hibernate tanto en Gradle como en Maven:

```

<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>{version}</version>
  </dependency>
</dependencies>

```

Reemplazando `{version}` por la versión de Hibernate a utilizar.

Hibernate se divide en varios módulos/artefactos bajo el grupo `org.hibernate.orm`. El artefacto principal se llama `hibernate-core`. Algunos de los módulos se encuentran [aquí](#).

Hibernate también proporciona un módulo de plataforma (**BOM** en terminología Maven) que se puede utilizar para alinear las versiones de los módulos de Hibernate junto con las versiones de sus bibliotecas. El artefacto de la plataforma se denomina `hibernate-platform`.

Para utilizar la plataforma BOM en Maven:

```

<dependencies>
  <dependency>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-core</artifactId>
  </dependency>
  <dependency>
    <groupId>jakarta.transaction</groupId>
    <artifactId>jakarta.transaction-api</artifactId>
  </dependency>
</dependencies>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.hibernate.orm</groupId>
      <artifactId>hibernate-platform</artifactId>
      <version>6.5.2.Final</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

```

También se debe agregar una dependencia para el controlador JDBC de la base de datos utilizada:

- PostgreSQL or CockroachDB -> `org.postgresql:postgresql:{version}`
- MySQL or TiDB -> `com.mysql:mysql-connector-j:{version}`
- MariaDB -> `org.mariadb.jdbc:mariadb-java-client:{version}`
- DB2 -> `com.ibm.db2:jcc:{version}`
- SQL Server -> `com.microsoft.sqlserver:mssql-jdbc:${version}`
- Oracle -> `com.oracle.database.jdbc:ojdbc11:${version}`
- H2 -> `com.h2database:h2:{version}`
- HSQLDB -> `org.hsqldb:hsqldb:{version}`

Donde `{versión}` es la última versión del controlador JDBC para la base de datos.

```
<dependencies>
  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>2.2.224</version>
  </dependency>
</dependencies>
```

Optional dependencies

Opcionalmente, también se puede agregar dependencias opcionales para trabajar con Hibernate como por ejemplo [SLF4J](#) o un [Query Validator](#) para comprobar las sentencias HQL en tiempo de compilación.

La lista está disponible en la documentación oficial.

Configuration using JPA XML

Siguiendo el enfoque estándar JPA, se proporcionaría un archivo llamado `persistence.xml`, que generalmente se coloca en el directorio `META-INF` de un archivo de persistencia, es decir, del archivo o directorio `.jar` que contiene las clases de entidad.

```
<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
    https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
  version="2.0">

  <persistence-unit name="org.hibernate.example">
    <class>org.hibernate.example.Book</class>
    <class>org.hibernate.example.Author</class>
    <properties>
      <!-- PostgreSQL -->
      <property name="jakarta.persistence.jdbc.url"
        value="jdbc:postgresql://localhost/example"/>

      <!-- Credentials -->
      <property name="jakarta.persistence.jdbc.user" value="gavin"/>
      <property name="jakarta.persistence.jdbc.password" value="hibernate"/>

      <!-- Automatic schema export -->
      <property name="jakarta.persistence.schema-generation.database.action"
        value="drop-and-create"/>

      <!-- SQL statement logging -->
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.format_sql" value="true"/>
      <property name="hibernate.highlight_sql" value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

Cada elemento `<class>` especifica el nombre completo de una clase de entidad.

En algunos entornos de contenedores, por ejemplo, en cualquier contenedor EE, los elementos `<class>` son innecesarios, ya que el contenedor escaneará el archivo en busca de clases anotadas y reconocerá automáticamente cualquier clase anotada como `@Entity`.

Cada elemento `<property>` especifica una propiedad de configuración y su valor. En el ejemplo hay propiedades definidas en el estándar de JPA (`'jakarta.persistence'`) y otras propiedades son específicas de Hibernate (`'hibernate'`).

Se puede obtener una `EntityManagerFactory` llamando a `Persistence.createEntityManagerFactory()` :

```
EntityManagerFactory entityManagerFactory =
    Persistence.createEntityManagerFactory("org.hibernate.example");
```

Si fuera necesario, es posible anular las propiedades de configuración especificadas en el fichero `persistence.xml` :

```
EntityManagerFactory entityManagerFactory =
    Persistence.createEntityManagerFactory("org.hibernate.example",
        Map.of(AvailableSettings.JAKARTA_JDBC_PASSWORD, password));
```

Configuration using Hibernate API

Alternativamente, la venerable clase `Configuration` permite configurar una instancia de Hibernate en código Java:

```
SessionFactory sessionFactory =
    new Configuration()
        .addAnnotatedClass(Book.class)
        .addAnnotatedClass(Author.class)
        // PostgreSQL
        .setProperty(AvailableSettings.JAKARTA_JDBC_URL, "jdbc:postgresql://localhost/example")
        // Credentials
        .setProperty(AvailableSettings.JAKARTA_JDBC_USER, user)
        .setProperty(AvailableSettings.JAKARTA_JDBC_PASSWORD, password)
        // Automatic schema export
        .setProperty(AvailableSettings.JAKARTA_HBM2DDL_DATABASE_ACTION,
            Action.SPEC_ACTION_DROP_AND_CREATE)
        // SQL statement logging
        .setProperty(AvailableSettings.SHOW_SQL, true)
        .setProperty(AvailableSettings.FORMAT_SQL, true)
        .setProperty(AvailableSettings.HIGHLIGHT_SQL, true)
        // Create a new SessionFactory
        .buildSessionFactory();
```

La clase `Configuration` ha sobrevivido casi sin cambios desde las primeras versiones (anteriores a la 1.0) de Hibernate, por lo que no parece particularmente moderna. Por otro lado, es muy fácil de usar y expone algunas opciones que `persistence.xml` no admite.

En realidad, la clase `Configuration` es solo una fachada muy simple para la API más moderna y más poderosa (pero más compleja) definida en el paquete `org.hibernate.boot` . Esta API es útil si se tienen requisitos avanzados, por ejemplo, en el desarrollo de un framework o un contenedor.

Configuration using Hibernate properties file

Si se utiliza la API `Configuration` de Hibernate, pero se quiere evitar el uso de ciertas propiedades de configuración directamente en el código Java, se pueden especificar en un archivo llamado `hibernate.properties` y colocar el archivo en la ruta de clase raíz:

```
# PostgreSQL
jakarta.persistence.jdbc.url=jdbc:postgresql://localhost/example
# Credentials
jakarta.persistence.jdbc.user=hibernate
jakarta.persistence.jdbc.password=zAh7mY42MNshzAQ5

# SQL statement logging
hibernate.show_sql=true
```

```
hibernate.format_sql=true
hibernate.highlight_sql=true
```

Basic configuration settings

La clase `AvailableSettings` enumera todas las propiedades de configuración que entiende Hibernate.

La lista es extensa, sin embargo la mayoría de estas propiedades rara vez se necesitan y muchas solo existen para brindar compatibilidad con versiones anteriores de Hibernate. Con raras excepciones, el comportamiento predeterminado de cada una de estas propiedades es el recomendable.

Las propiedades que deben configurarse para empezar son:

- `jakarta.persistence.jdbc.url`: la URL JDBC de la base de datos
- `jakarta.persistence.jdbc.user`: credenciales de acceso a la base de datos
- `jakarta.persistence.jdbc.password`: credenciales de acceso a la base de datos

En términos de optimización de rendimiento, también es recomendable configurar la propiedad `hibernate.connection.pool_size`.

Automatic schema export

Esta característica de Hibernate es una característica que permite a Hibernate **generar y exportar automáticamente el esquema de la base de datos** basado en las entidades mapeadas en la aplicación. Esto puede incluir la creación, actualización, o eliminación de tablas, índices y llaves foráneas, entre otros elementos del esquema de la base de datos.

Puede hacer que Hibernate infiera el esquema de su base de datos a partir de las anotaciones de mapeo que ha especificado en su código Java, y exportar el esquema al momento de la inicialización especificando una o más de las siguientes propiedades de configuración:

- `jakarta.persistence.schema-generation.database.action`
- `jakarta.persistence.create-database-schemas` (opcional)
- `jakarta.persistence.schema-generation.create-source` (opcional)
- `jakarta.persistence.schema-generation.create-script-source` (opcional)
- `jakarta.persistence.sql-load-script-source` (opcional)

Esta característica es extremadamente útil para realizar pruebas. Durante las fases iniciales de desarrollo o cuando se está creando un prototipo, es útil tener Hibernate manejando automáticamente el esquema.

La forma más sencilla de preinicializar una base de datos con datos de prueba o "de referencia" es colocar una lista de declaraciones SQL de inserción en un archivo llamado, por ejemplo, *'import.sql'*, y especificar la ruta a este archivo utilizando la propiedad `jakarta.persistence.sql-load-script-source`.

Este enfoque es más limpio que escribir código Java para instanciar entidades y llamar a `persist()` en cada una de ellas.

Logging the generated SQL

Para ver el SQL generado mientras se envía a la base de datos, tiene dos opciones.

Una forma es establecer la propiedad `hibernate.show_sql=true`, y Hibernate registrará el SQL directamente en la consola. Puede hacer que la salida sea mucho más legible habilitando el formateo o el resaltado. Estas configuraciones son muy útiles para solucionar problemas con las declaraciones SQL generadas:

- **hibernate.show_sql**: si es `'true'`, imprime el registro SQL directamente en la consola
- **hibernate.format_sql**: si es `'true'`, imprime el registro SQL en un formato con sangría de varias líneas
- **hibernate.highlight_sql**: si es `'true'`, imprime el registro SQL con resaltado de sintaxis mediante códigos de escape ANSI

Como alternativa, puede habilitar el registro en nivel de depuración (debug) para la categoría `org.hibernate.SQL` utilizando su implementación preferida de logging **SLF4J**.

Minimizing repetitive mapping information

Las propiedades `hibernate.default_schema` o `hibernate.default_catalog` son muy útiles para minimizar la cantidad de información que necesitará especificar explícitamente en las anotaciones `@Table` y `@Column`. [Más información](#)

Escribir su propia `PhysicalNamingStrategy` y/o `ImplicitNamingStrategy` es una forma especialmente buena de reducir el desorden de las anotaciones en las clases de entidad, e implementar sus convenciones de nomenclatura de base de datos.

Por lo tanto, se debería hacer para cualquier modelo de datos que no sea trivial. [Más información](#)

Nationalized character data in SQL Server

Por defecto, los tipos `'char'` y `'varchar'` de **SQL Server** no admiten datos Unicode. Sin embargo, una cadena de Java puede contener cualquier carácter Unicode. Por lo tanto, si está trabajando con SQL Server, es posible que necesite forzar a Hibernate a usar los tipos de columna `'nchar'` y `'nvarchar'`:

- **hibernate.use_nationalized_character_data**: `'nchar'` y `'nvarchar'` en lugar de `'char'` y `'varchar'`


Por otro lado, si solo algunos campos almacenan datos nacionalizados, utilice la anotación `@Nationalized` para indicar los campos de sus entidades que mapean estas columnas.

Como alternativa, puede configurar SQL Server para usar la intercalación habilitada para UTF-8 (`_UTF8`).

Entities

TODO

Enlaces de interés

-  <https://hibernate.org>
- <https://hibernate.org/orm/documentation/6.5>
- <https://hibernate.org/orm/documentation/getting-started>
- <https://www.baeldung.com/tag/hibernate>

Licencia



Esta obra está bajo una [licencia de Creative Commons Reconocimiento-Compartir Igual 4.0 Internacional](#).