Getting Started with Hibernate

Version 6.4.0-SNAPSHOT

Table of Contents

Preface

Hibernate is an *Object/Relational Mapping* (ORM) solution for programs written in Java and other JVM languages.

While a strong background in SQL is not required to use Hibernate, a basic understanding of its concepts is useful - especially the principles of *data modeling*. Understanding the basics of transactions and design patterns such as *Unit of Work* are important as well.

Useful background resources

- Data Modeling (Wikipedia).
- Data Modeling 101
- Java & Databases: An Overview of Libraries & APIs
- Unit of Work

Chapter 1. Obtaining Hibernate

Hibernate is broken into a number of modules/artifacts under the `org.hibernate.orm` group. The main artifact is named `hibernate-core`.

This guide uses 6.4.0-SNAPSHOT as the Hibernate version for illustration purposes. Be sure to change this version, if necessary, to the version you wish to use.

We can declare a dependency on this artifact using Gradle

```
dependencies {
  implementation "org.hibernate.orm:hibernate-core:6.4.0-SNAPSHOT"
}
```

or Maven:

```
<dependency>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>6.4.0-SNAPSHOT</version>
</dependency>
```

1.1. Hibernate ORM modules

As mentioned earlier, Hibernate ORM is broken into a number of modules with the intent of isolating transitive dependencies based on the features being used or not.

Table 1. API-oriented modules

`hibernate-core`	The core object/relational mapping engine
`hibernate-envers`	Entity versioning and auditing
`hibernate-spatial`	Support for spatial/GIS data types using GeoLatte
`hibernate-jpamodelgen`	An annotation processor that generates a JPA-compliant metamodel, plus optional Hibernate extras

Table 2. Integration-oriented modules

`hibernate-agroal`	Support for Agroal connection pooling
`hibernate-c3p0`	Support for C3P0 connection pooling
`hibernate-hikaricp`	Support for HikariCP connection pooling
`hibernate-vibur`	Support for Vibur DBCP connection pooling
`hibernate-proxool`	Support for Proxool connection pooling

`hibernate-jcache`	Integration with JCache, allowing any compliant implementation as a second-level cache provider
`hibernate-graalvm`	Experimental extension to make it easier to compile applications as a GraalVM native image
`hibernate-micrometer`	Integration with Micrometer metrics
`hibernate-community-dialects`	Additional community-supported SQL dialects

Table 3. Testing-oriented modules

`hibernate-testing`	A series of JUnit extensions for testing Hibernate ORM functionality

1.2. Platform / BOM

Hibernate also provides a platform (BOM in Maven terminology) module which can be used to align versions of the Hibernate modules along with the versions of its libraries. The platform artifact is named `hibernate-platform`.

To apply the platform in Gradle

```
dependencies {
  implementation platform "org.hibernate.orm:hibernate-platform:6.4.0-SNAPSHOT"

  // use the versions from the platform
  implementation "org.hibernate.orm:hibernate-core"
  implementation "jakarta.transaction:jakarta.transaction-api"
}
```

See the Gradle documentation for capabilities of applying a platform.

To apply the platform (BOM) in Maven

```
<dependency>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-core</artifactId>
</dependency>
<dependency>
    <groupId>jakarta.transaction</groupId>
    <artifactId>jakarta.transaction-api</artifactId>
</dependency>

<dependencyManagement>
  <dependency>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-platform</artifactId>
    <version>6.4.0-SNAPSHOT</version>
    <type>pom</type>
```

```
    <scope>import</scope>
  </dependency>
</dependencyManagement>
```

1.3. Example sources

The bundled examples mentioned in this tutorial can be downloaded from here.

Alternatively, the example source code can also be obtained from Github

Chapter 2. Tutorial using native Hibernate APIs

Objectives

- ☑ Configure Hibernate using `hibernate.properties`
- ☑ Create a `SessionFactory` using native bootstrapping
- ☑ Use annotations to provide mapping information
- ☑ Use `Session` to persist and query data

This tutorial is located within the download bundle under `annotations/`.

2.1. Configuration via properties file

In this example, configuration properties are specified in a file named `hibernate.properties`.

Configuration via `hibernate.properties`

```
# Database connection settings
hibernate.connection.url=jdbc:h2:mem:db1;DB_CLOSE_DELAY=-1
hibernate.connection.username=sa
hibernate.connection.password=

# Echo all executed SQL to console
hibernate.show_sql=true
hibernate.format_sql=true
hibernate.highlight_sql=true

# Automatically export the schema
hibernate.hbm2ddl.auto=create
```

The following properties specify JDBC connection information:

Table 4. JDBC connection settings

Configuration property name	Purpose
`jakarta.persistence.jdbc.url`	JDBC URL of your database
`jakarta.persistence.jdbc.user` and `jakarta.persistence.jdbc.password`	Your database credentials

These tutorials use the H2 embedded database, so the values of these properties are specific to running H2 in its in-memory mode.

These properties enable logging of SQL to the console as it is executed, in an aesthetically pleasing format:

Table 5. Settings for SQL logging to the console

Configuration property name	Purpose
`hibernate.show_sql`	If `true`, log SQL directly to the console
`hibernate.format_sql`	If `true`, log SQL in a multiline, indented format
`hibernate.highlight_sql`	If `true`, log SQL with syntax highlighting via ANSI escape codes

When developing persistence logic with Hibernate, it's very important to be able to see exactly what SQL is being executed.

2.2. The annotated entity Java class

The entity class in this tutorial is `org.hibernate.tutorial.annotations.Event`. Observe that:

- This class uses standard JavaBean naming conventions for property getter and setter methods, as well as private visibility for the fields. This is recommended, but it's not a requirement.
- The no-argument constructor, which is also a JavaBean convention, *is* a requirement for all persistent classes. Hibernate needs to instantiate objects for you, using Java Reflection. The constructor should have package-private or `public` visibility, to allow Hibernate to generate proxies and optimized code for field access.

The Entity types section of the User Guide covers the complete set of requirements for the entity class.

We use annotations to identify the class as an entity, and to map it to the relational schema.

Identifying the class as an entity

```
@Entity   ①
@Table(name = "Events")   ②
public class Event {
    ...
}
```

① `@jakarta.persistence.Entity` marks the `Event` class as an entity.

② `@jakarta.persistence.Table` explicitly specifies the name of the mapped table. Without this annotation, the table name would default to `Event`.

Every entity class must have an identifier.

Identifying the identifier property

```
@Id   ①
@GeneratedValue   ②
```

```
private Long id;
```

① `@jakarta.persistence.Id` marks the field as holding the identifier (primary key) of the entity.

② `@jakarta.persistence.GeneratedValue` specifies that this is a *synthetic id*, that is, a system-generated identifier (a surrogate primary key).

Other fields of the entity are considered persistent by default.

Mapping basic properties

```
private String title;

@Column(name = "eventDate")   ①
private LocalDateTime date;
```

① `@jakarta.persistence.Column` explicitly specifies the name of a mapped column. Without this annotation, the column name would default to `date`, which is a keyword on some databases.

2.3. Example code

The class `org.hibernate.tutorial.annotations.HibernateIllustrationTest` illustrates the use of the Hibernate's native APIs, including:

- `Session` and `SessionFactory`, and
- `org.hibernate.boot` for configuration and bootstrap.

There are several different ways to configure and start Hibernate, and this is not even the most common approach.

The examples in these tutorials are presented as JUnit tests. A benefit of this approach is that `setUp()` and `tearDown()` roughly illustrate how a `org.hibernate.SessionFactory` is created when the program starts, and closed when the program terminates.

Obtaining the `SessionFactory`

```
protected void setUp() {
    // A SessionFactory is set up once for an application!
    final StandardServiceRegistry registry =
            new StandardServiceRegistryBuilder()
                    .build();   ① ②
    try {
        sessionFactory =
                new MetadataSources(registry)              ③
                        .addAnnotatedClass(Event.class)    ④
                        .buildMetadata()                   ⑤
                        .buildSessionFactory();            ⑥
    }
    catch (Exception e) {
```

```
        // The registry would be destroyed by the SessionFactory, but we
        // had trouble building the SessionFactory so destroy it manually.
        StandardServiceRegistryBuilder.destroy(registry);
    }
}
```

① The `setUp()` method first builds a `StandardServiceRegistry` instance which incorporates configuration information into a working set of `Services` for use by the `SessionFactory`.

② Here we put all configuration information in `hibernate.properties`, so there's not much interesting to see.

③ Using the `StandardServiceRegistry` we create the `MetadataSources` which lets us tell Hibernate about our domain model.

④ Here we have only one entity class to register.

⑤ An instance of `Metadata` represents a complete, partially-validated view of the application domain model.

⑥ The final step in the bootstrap process is to build a `SessionFactory` for the configured services and validated domain model. The `SessionFactory` is a thread-safe object that's instantiated once to serve the entire application.

The `SessionFactory` produces instances of `Session`. Each session should be thought of as representing a *unit of work*.

Persisting entities

```
sessionFactory.inTransaction(session -> {   ①
    session.persist(new Event("Our very first event!", now()));   ②
    session.persist(new Event("A follow up event", now()));
});
```

① The `inTransaction()` method creates a session and starts a new transaction.

② Here we create two new `Event` objects and hands them over to Hibernate, calling the `persist()` method to make these instances persistent. Hibernate is responsible for executing an `INSERT` statement for each `Event`.

Obtaining a list of entities

```
sessionFactory.inTransaction(session -> {
    session.createSelectionQuery("from Event", Event.class)   ①
            .getResultList()   ②
            .forEach(event -> out.println("Event (" + event.getDate() + ") : " +
event.getTitle()));
});
```

① Here we use a very simple *Hibernate Query Language* (HQL) statement to load all existing `Event` objects from the database.

② Hibernate generates and executes the appropriate `SELECT` statement, and then instantiates and populates `Event` objects with the data in the query result set.

2.4. Take it further!

Practice Exercises

- ☐ Actually run this example to see the SQL executed by Hibernate displayed in the console.
- ☐ Reconfigure the examples to connect to your own persistent relational database.
- ☐ Add an association to the `Event` entity to model a message thread.

Chapter 3. Tutorial using JPA-standard APIs

Objectives

- ☑ Configure Hibernate using `peristence.xml`
- ☑ Bootstrap a Jakarta Persistence `EntityManagerFactory`
- ☑ Use annotations to provide mapping information
- ☑ Use `EntityManager` to persist and query data

This tutorial is located within the download bundle under `entitymanager/`.

3.1. persistence.xml

JPA defines a different bootstrap process, along with a standard configuration file format named `persistence.xml`. In Java™ SE environments the persistence provider (Hibernate) is required to locate every JPA configuration file in the classpath at the path `META-INF/persistence.xml`.

Configuration via `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="org.hibernate.tutorial.jpa">   ①
        <description>
            Persistence unit for the Jakarta Persistence tutorial of the Hibernate
Getting Started Guide
        </description>

        <class>org.hibernate.tutorial.em.Event</class>     ②

        <properties>    ③
            <!-- Database connection settings -->
            <property name="jakarta.persistence.jdbc.url"
value="jdbc:h2:mem:db1;DB_CLOSE_DELAY=-1" />
            <property name="jakarta.persistence.jdbc.user" value="sa" />
            <property name="jakarta.persistence.jdbc.password" value="" />

            <!-- Automatically export the schema -->
            <property name="jakarta.persistence.schema-generation.database.action"
value="create" />

            <!-- Echo all executed SQL to console -->
            <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>
```

① A `persistence.xml` file should provide a unique name for each *persistence unit* it declares. Applications use this name to reference the configuration when obtaining an `EntityManagerFactory` as we will see shortly.

② The `<class/>` element registers our annotated entity class.

③ The settings specified as `<properties/>` elements were already discussed in Configuration via properties file. Here JPA-standard property names are used where possible.

Configuration properties prefixed with the legacy Java EE namespace `javax.persistence` are still recognized, but the Jakarta EE namespace `jakarta.persistence` should be preferred.

3.2. The annotated entity Java class

The entity class is exactly the same as in The annotated entity Java class.

3.3. Example code

The previous tutorials used Hibernate native APIs. This tutorial uses the standard Jakarta Persistence APIs.

Obtaining the JPA EntityManagerFactory

```
protected void setUp() {
    entityManagerFactory = Persistence.createEntityManagerFactory
("org.hibernate.tutorial.jpa");   ①
}
```

① Notice again that the persistence unit name is `org.hibernate.tutorial.jpa`, which matches the name from our persistence.xml.

The code to persist and query entities is almost identical to Persisting entities. Unfortunately, `EntityManagerFactory` doesn't have a nice `inTransaction()` method like `SessionFactory` does, so we had to write our own:

Managing transactions in JPA

```
void inTransaction(Consumer<EntityManager> work) {
    EntityManager entityManager = entityManagerFactory.createEntityManager();
    EntityTransaction transaction = entityManager.getTransaction();
    try {
        transaction.begin();
```

```
            work.accept(entityManager);
            transaction.commit();
        }
        catch (Exception e) {
            if (transaction.isActive()) {
                transaction.rollback();
            }
            throw e;
        }
        finally {
            entityManager.close();
        }
    }
```

If you use JPA in Java SE, you'll need to copy/paste this function into your project. Alternatively you could unwrap the `EntityManagerFactory` as a `SessionFactory`.

3.4. Take it further!

Practice Exercises

- ☐ Learn how to use CDI to inject a container-managed `EntityManager` in Quarkus. See the Quarkus website for instructions.

Chapter 4. Tutorial Using Envers

Objectives

- ☑ Annotate an entity as historical
- ☑ Configure Envers
- ☑ Use the Envers APIs to view and analyze historical data

> This tutorial is located within the download bundle under `envers/`.

4.1. persistence.xml

This file is unchanged from what we had before.

4.2. The annotated entity Java class

The entity class is also almost identical to what we had previously. The major difference is the addition of the annotation `@org.hibernate.envers.Audited`, which tells Envers to automatically track changes to this entity.

4.3. Example code

The code saves some entities, makes a change to one of the entities and then uses the Envers API to pull back the initial revision as well as the updated revision. A revision refers to a historical snapshot of an entity.

Using the `org.hibernate.envers.AuditReader`

```
public void testBasicUsage() {
    ...
    AuditReader reader = AuditReaderFactory.get( entityManager );   ①
    Event firstRevision = reader.find( Event.class, 2L, 1 );        ②
    ...
    Event secondRevision = reader.find( Event.class, 2L, 2 );       ③
    ...
}
```

① An `org.hibernate.envers.AuditReader` is obtained from the `org.hibernate.envers.AuditReaderFactory` which wraps the JPA `EntityManager`.

② The `find` method retrieves specific revisions of the entity. The first call retrieves revision number 1 of the `Event` with id 2.

③ Later, the second call asks for revision number 2 of the `Event` with id 2.

4.4. Take it further!

Practice Exercises

- ☑ Provide a custom revision entity to additionally capture who made the changes.
- ☑ Write a query to retrieve only historical data which meets some criteria. Use the *User Guide* to see how Envers queries are constructed.
- ☑ Experiment with auditing entities which have various forms of relationships (many-to-one, many-to-many, etc). Try retrieving historical versions (revisions) of such entities and navigating the object tree.

Chapter 5. Credits

The full list of contributors to Hibernate ORM can be found on the GitHub repository.

The following contributors were involved in this documentation:

- Steve Ebersole