503111

Java Technology

HIBERNATE

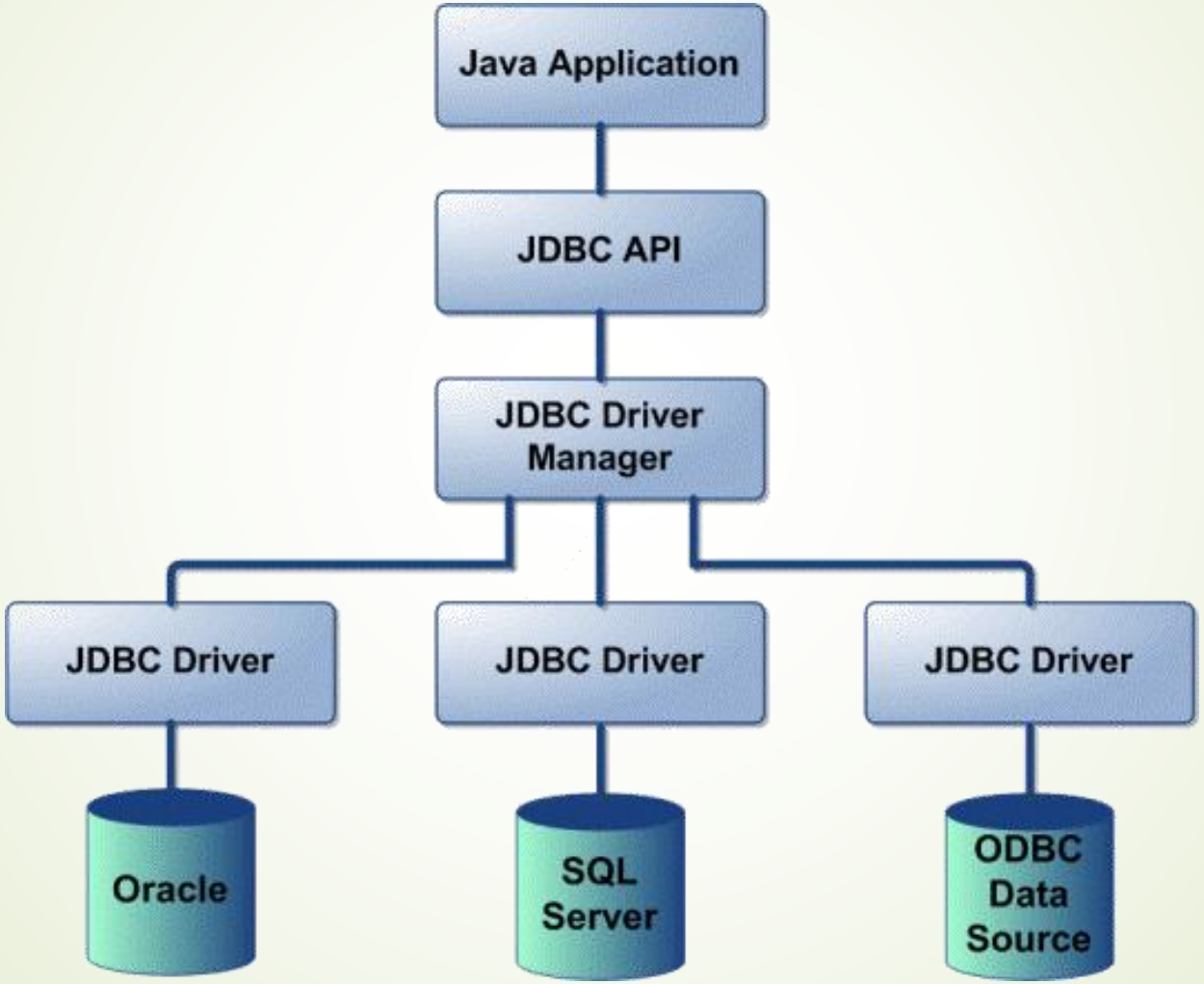
Outline

1. Hibernate Overview
2. Environment Setup
3. Configurations
4. Entity & Mapping
5. Hibernate Query Language
6. Relationships

What is JDBC?

- JDBC stands for **Java Database Connectivity.**
- It provides a set of Java API for accessing the relational databases from Java program.
- These Java APIs enables Java programs to execute SQL statements and interact with any SQL compliant database.

What is JDBC?


Java Application
JDBC API
JDBC Driver Manager
JDBC Driver
JDBC Driver
JDBC Driver
Oracle
SQL Server
ODBC Data Source

JDBC Pros and Cons

- Pros:
  1. Clean and simple SQL processing
  2. Good performance with large data
  3. Very good for small applications
  4. Simple syntax so easy to learn
- Cons:
  1. Complex if it is used in large projects
  2. Large programming overhead
  3. No encapsulation
  4. Hard to implement MVC concept
  5. Query is DBMS specific

Java ORM Frameworks

- There are several persistent frameworks and ORM options in Java.
- A persistent framework is an ORM service that stores and retrieves objects into a relational database.

1. Enterprise JavaBeans Entity Beans
2. Java Data Objects
3. Castor
4. TopLink
5. Spring DAO
6. Hibernate
7. And many more

Hibernate Overview

- Hibernate is an Object-Relational Mapping (ORM) solution for JAVA.
- Hibernate sits between traditional Java objects and database server to handle all the works in persisting those objects based on the appropriate O/R mechanisms and patterns.

Hibernate Advantages

- Hibernate takes care of mapping Java classes to database tables using XML files and without writing any line of code.
- Provides simple APIs for storing and retrieving Java objects directly to and from the database.
- If there is change in the database or in any table, then you need to change the XML file properties only.
- Abstracts away the unfamiliar SQL types and provides a way to work around familiar Java Objects.
- Hibernate does not require an application server to operate.
- Manipulates Complex associations of objects of your database.
- Minimizes database access with smart fetching strategies.
- Provides simple querying of data.

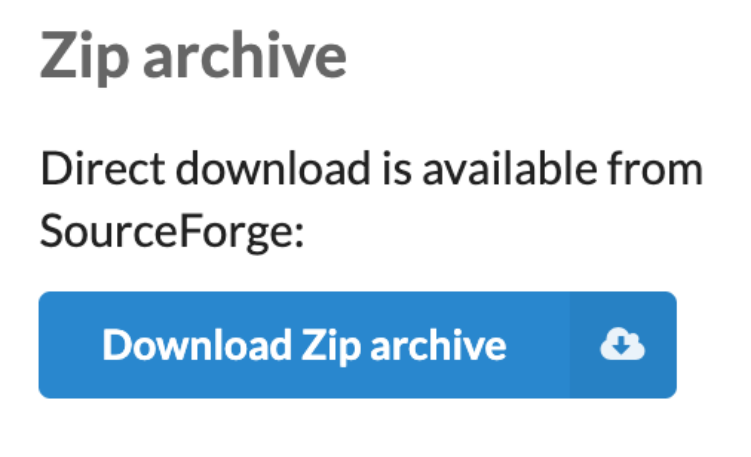
Environment Setup

- System Requirements:
  - Hibernate 5.2 and later versions: at least Java 1.8 and JDBC 4.2.
  - Hibernate 5.1 and older versions: at least Java 1.6 and JDBC 4.0.
- Hibernate with Maven:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.5.3.Final</version>
</dependency>
```

Environment Setup

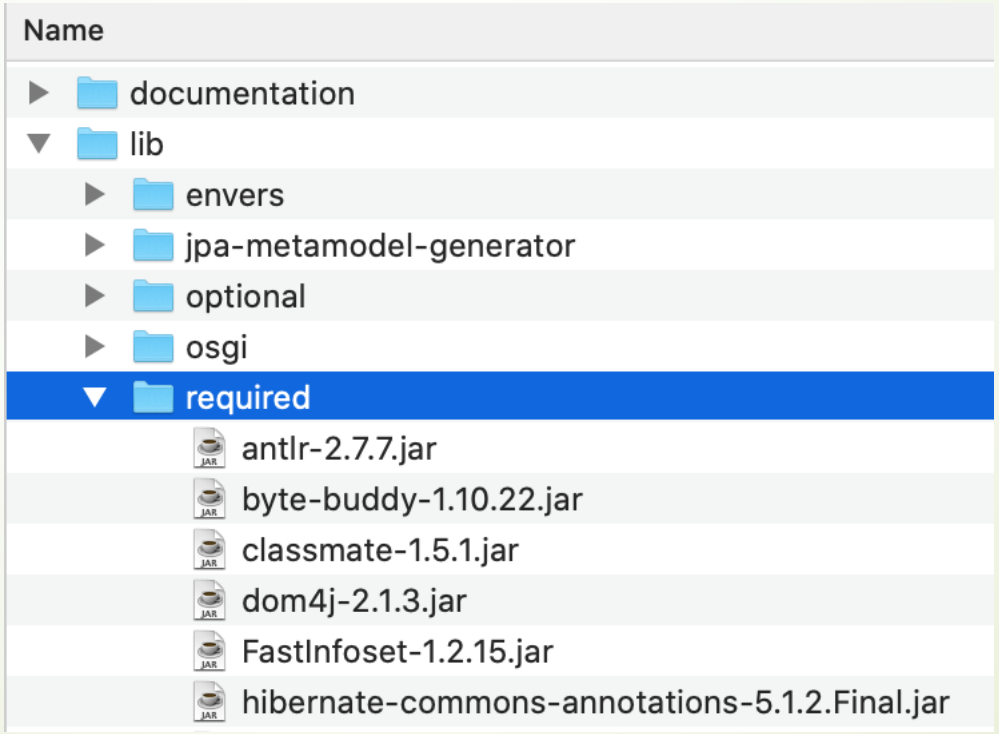
- Traditional approach:
  1. Download Hibernate at https://hibernate.org/orm/releases/5.4/



Environment Setup

- Traditional approach:

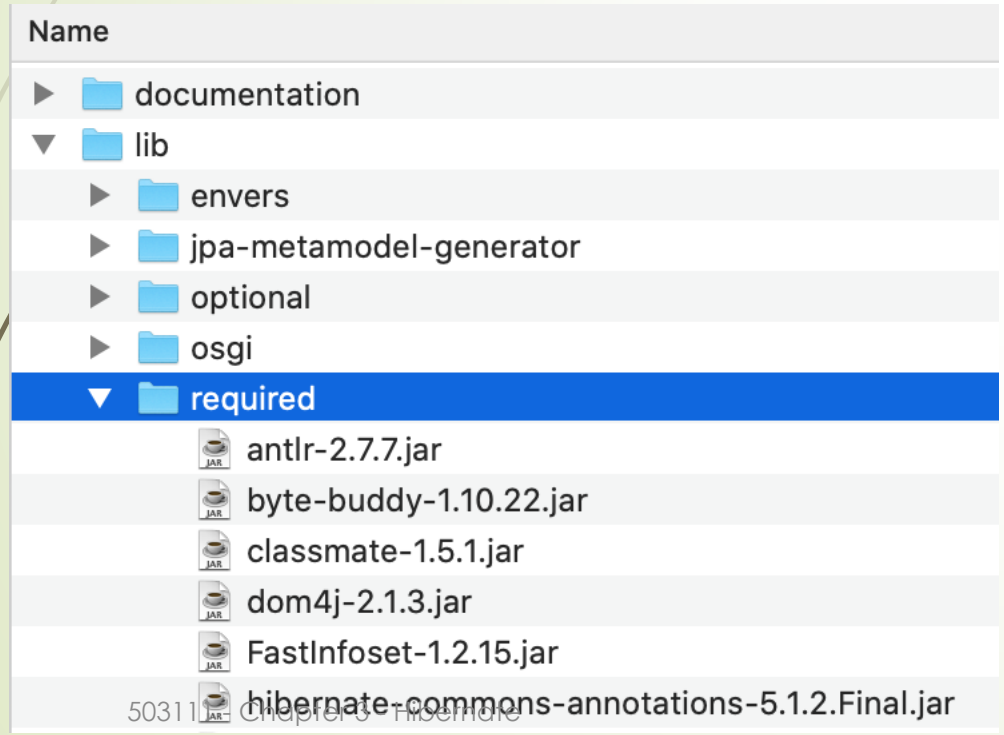
  2. Extract the hibernate-release-5.4.32.Final.zip file to a specific location

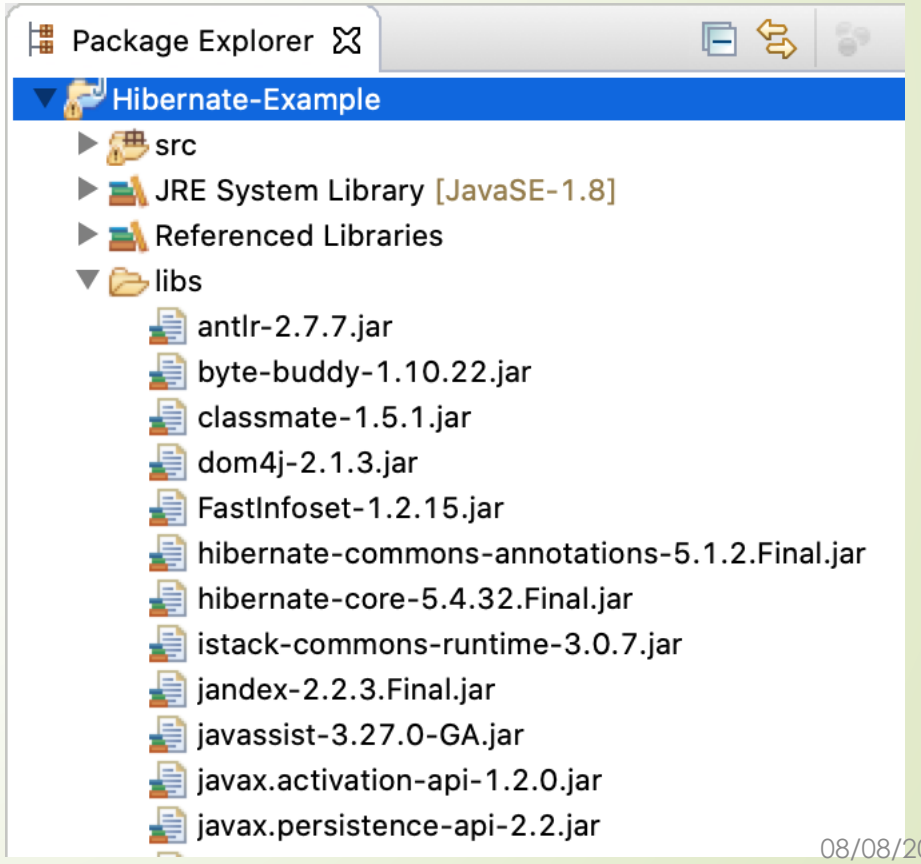


Environment Setup

- Traditional approach:

  3. Copy all the *.jar file from the required folder to the class path of java project





Configurations

- Hibernate requires to know in advance: where to find the mapping information that defines how your Java classes relate to the database tables.
- All such information is usually supplied as an XML file named hibernate.cfg.xml.
- This configuration file is placed at the **src** folder in the project.

Configurations

hibernate.cfg.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<hibernate-configuration>
    <session-factory>
        <property
  name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="hibernate.connection.password">123456</property>
        <property name="hibernate.connection.url">jdbc:mysql://127.0.0.1/Lab06</property>
        <property name="hibernate.connection.username">mvmanh</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>
        <property name="hibernate.current_session_context_class">thread</property>
        <property name="hibernate.hbm2ddl.auto">update</property>
        <property name="show_sql">true</property>

    </session-factory>
</hibernate-configuration>

```

Hibernate Properties

- Following is the list of important properties, you will be required to configure for a databases:
  - **hibernate.dialect**
    - This property makes Hibernate generate the appropriate SQL for the chosen database.
    - Ex: org.hibernate.dialect.MySQL8Dialect
  - **hibernate.connection.driver_class**
    - The JDBC driver class.
    - Ex: com.mysql.cj.jdbc.Driver
  - **hibernate.connection.url**
    - The JDBC URL to the database instance.
    - Ex: jdbc:mysql://127.0.0.1/Lab06

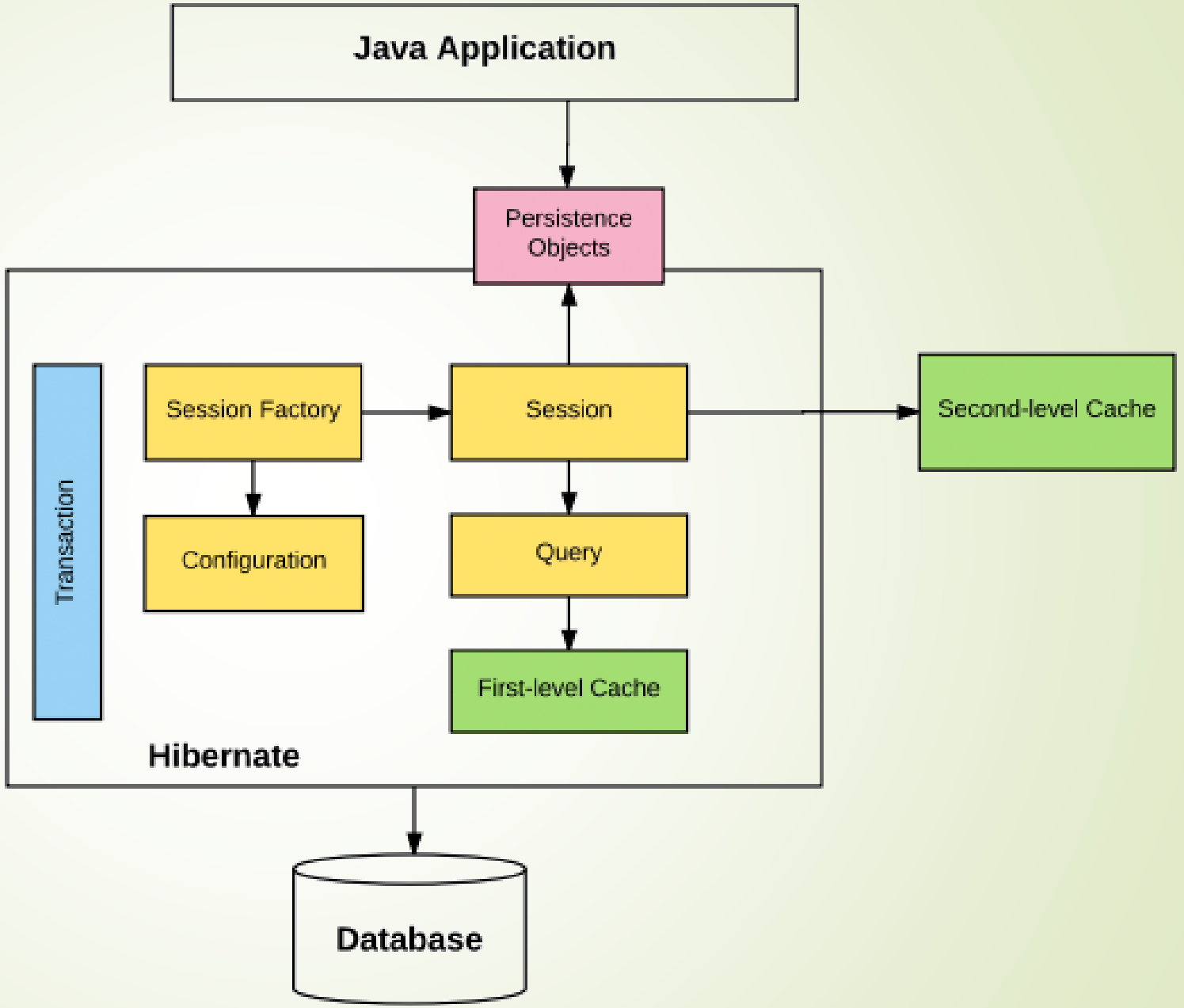
Hibernate Properties

- **hibernate.connection.username**
  - The database username.
- **hibernate.connection.password**
  - The database password.
- **hibernate.connection.pool_size**
  - Limits the number of connections waiting in the Hibernate database connection pool.
- **hibernate.connection.autocommit**
  - Allows autocommit mode to be used for the JDBC connection.

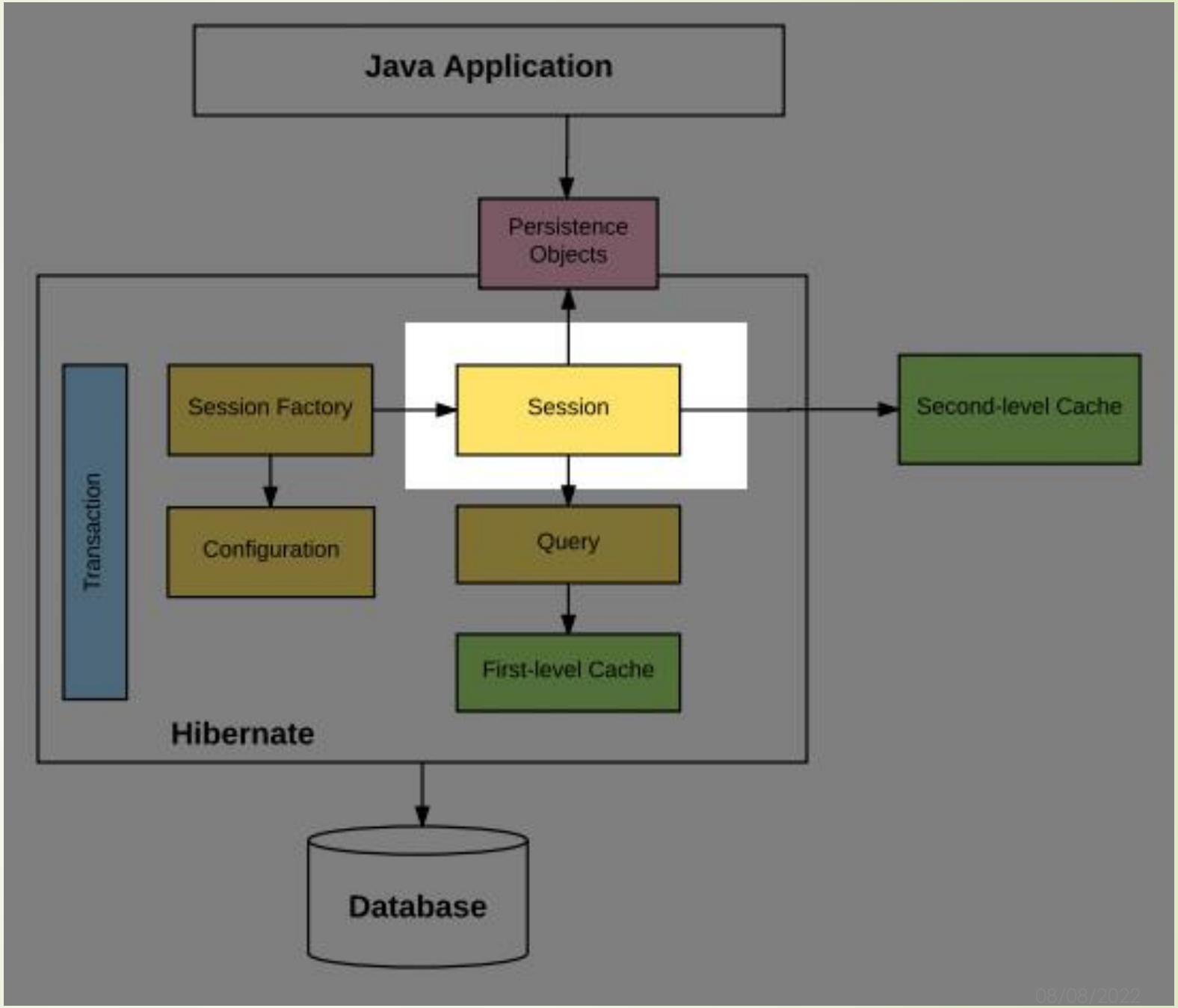
Database & Dialect Property

- **DB2**
  - org.hibernate.dialect.DB2Dialect
- **HSQLDB**
  - org.hibernate.dialect.HSQLDialect
- **MySQL**
  - org.hibernate.dialect.MySQLDialect
- **Progress**
  - org.hibernate.dialect.ProgressDialect
- **Microsoft SQL Server 2008**
  - org.hibernate.dialect.SQLServer2008Dialect

Session


Java Application
Persistence Objects
Transaction
Session Factory
Session
Second-level Cache
Configuration
Query
First-level Cache
Hibernate
Database

Session


Java Application
Persistence Objects
Session Factory
Session
Second-level Cache
Transaction
Configuration
Query
First-level Cache
Hibernate
Database

Session

- A Session is used to get a physical connection with a database.
- Persistent objects are saved and retrieved through a Session object.
- The session objects should not be kept open for a long time.
- Instances may exist in one of the following three states:
  1. **transient** – A new instance of a persistent class, which is not associated with a Session and has no representation in the database.
  2. **persistent** – A persistent instance has a representation in the database, an identifier value and is associated with a Session.
  3. **detached** – the persistent instance will become a detached instance when Once we close the Hibernate Session.

Persistent Classes

- Java classes whose objects will be stored in database tables are called persistent classes in Hibernate.
- Hibernate works best if these classes follow some simple rules, also known as the Plain Old Java Object (POJO) programming model.

Persistent Classes

```
1  public class Student {
2      private int id;
3      private String name;
4
5      public Student (int id, String name) {
6          this.id = id;
7          this.name = name;
8      }
9      // getter methods
10     // setter methods
11     // toString method
12 }
```

XML Mapping

- An Object/relational mappings are usually defined in an XML document.
- This mapping file instructs Hibernate — how to map the defined class or classes to the database tables.

```
1   <?xml version = "1.0" encoding = "utf-8"?>
2   <!DOCTYPE hibernate-mapping PUBLIC
3   "-//Hibernate/Hibernate Mapping DTD//EN"
4   "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
5   <hibernate-mapping>
6      <class name = "Student" table = "Student">
7         <id name = "id" type = "int" column = "id">
8            <generator class="native"/>
9         </id>
10        <property name = "name" column = "name" type = "string"/>
11     </class>
12  </hibernate-mapping>
```

Annotation Mapping

- Hibernate annotations are the newest way to define mappings without the use of XML file.
- You can use annotations in addition to or as a replacement of XML mapping metadata

Annotation Mapping

```java
import javax.persistence.*;

@Entity
@Table(name = "Student")
public class Student {

    @Id @GeneratedValue
    @Column(name = "id")
    private int id;

    @Column(name = "fullname")
    private String name;

    public Student (int id, String name) {
        this.id = id;
        this.name = name;
    }
}
```

Common Annotations

- @Entity: Specifies that the class is an entity. This annotation can be applied on Class, Interface of Enums.

```
1  import javax.persistence.Entity;
2
3  @Entity
4  public class Employee implements Serializable {
5      // content
6  }
```

Common Annotations

- @Table: it specifies the table in the database with which this entity is mapped.

```
1  import javax.persistence.Entity;
2
3  @Entity
4  @Table(name = "employee")
5  public class Employee implements Serializable {
6      // content
7  }
```

Common Annotations

- @Column: Specify the column mapping using @Column annotation.
  - Name attribute of this annotation is used for specifying the table's column name.

```
1  import javax.persistence.Entity;
2
3  @Entity
4  @Table(name = "employee")
5  public class Employee implements Serializable {
6
7      @Column(name = "employee_name")
8      private String employeeName;
9
10       //other contents
11 }
```

Common Annotations

- @ID: This annotation specifies the primary key of the entity.
- @GeneratedValue: This annotation specifies the generation strategies for the values of primary keys.

```
1  import javax.persistence.Entity;
2
3  @Entity
4  @Table(name = "employee")
5  public class Employee implements Serializable {
6
7      @Id
8      @Column(name = "id")
9      @GeneratedValue(strategy = GenerationType.IDENTITY)
10     private int id;
11
12     //other contents
13 }
```

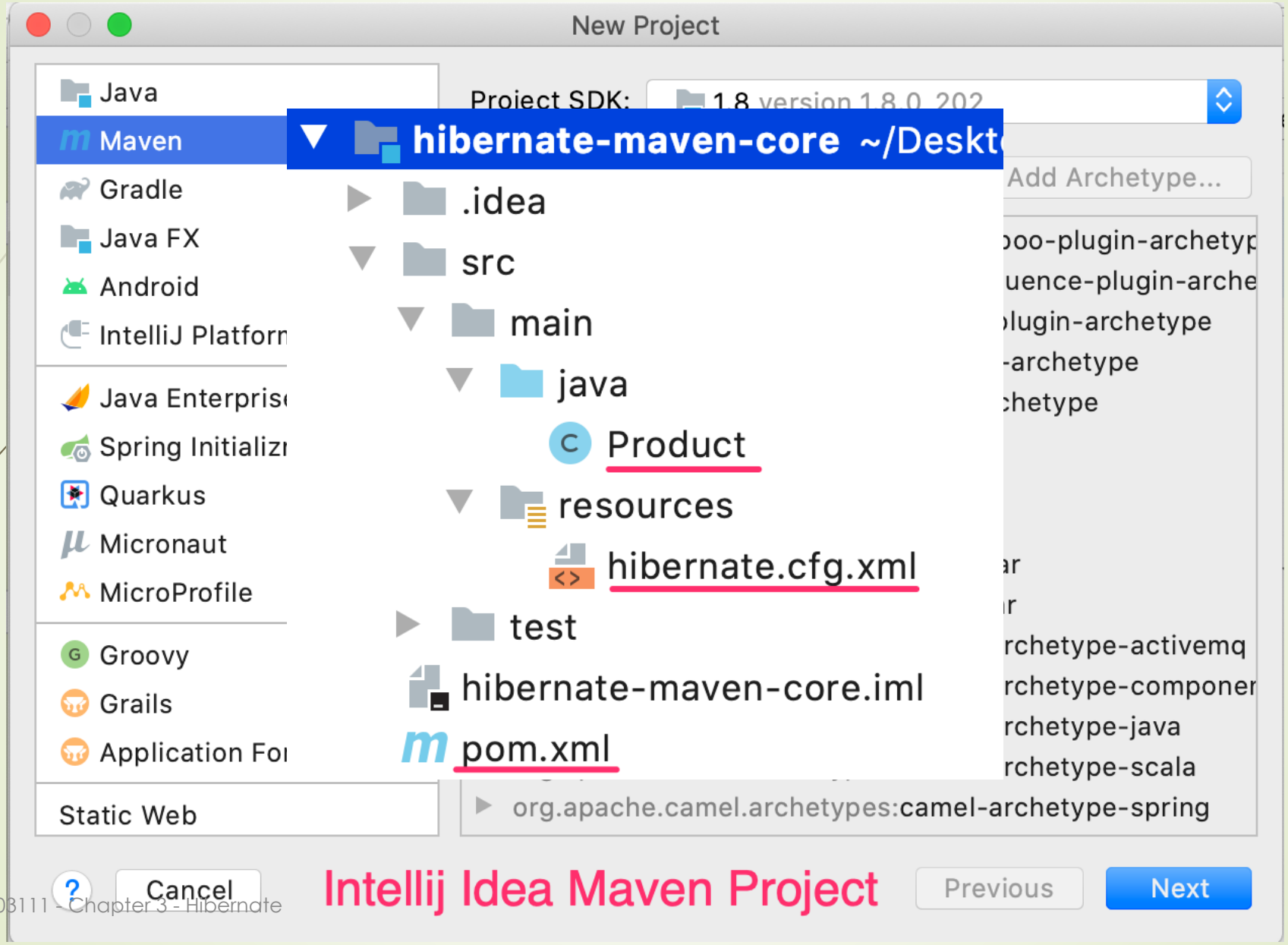
Hibernate Example

Hibernate Example

- This example will demonstrate all the steps required to:
  1. Setup the project
  2. Create a Product Entity
  3. Setup an instance of SessionFactory
  4. Create a session instance and save products to MySQL Database

Step 1. Setup the project

1. Create a maven project
2. Add Hibernate core dependency
3. Add Mysql connector dependency
4. Create hibernate.cfg.xml configuration file


New Project
Java
Maven
Gradle
Java FX
Android
IntelliJ Platform
Java Enterprise
Spring Initializr
Quarkus
Micronaut
MicroProfile
Groovy
Grails
Application For
Static Web
Project SDK:
1.8 version 1.8.0_202
Add Archetype...
hibernate-maven-core ~/Deskt
.idea
src
main
java
Product
resources
hibernate.cfg.xml
test
hibernate-maven-core.iml
pom.xml
org.apache.camel.archetypes:camel-archetype-spring
Cancel
Intellij Idea Maven Project
Previous
Next

```
<properties>
    <maven.compiler.source>8</maven.compiler.source>
    <maven.compiler.target>8</maven.compiler.target>
</properties>
```

pom.xml

```
<dependencies>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.20</version>
    </dependency>

    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.5.6.Final</version>
    </dependency>
</dependencies>
```

resources/hibernate.cfg.xml

```
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>
        <property name="hibernate.connection.url">jdbc:mysql://127.0.0.1/hibernate_demo</property>
        <property name="hibernate.connection.username">mvmanh</property>
        <property name="hibernate.connection.password">123456</property>
        <property name="show_sql">true</property>
        <property name="hibernate.hbm2ddl.auto">create</property>
    </session-factory>
</hibernate-configuration>
```

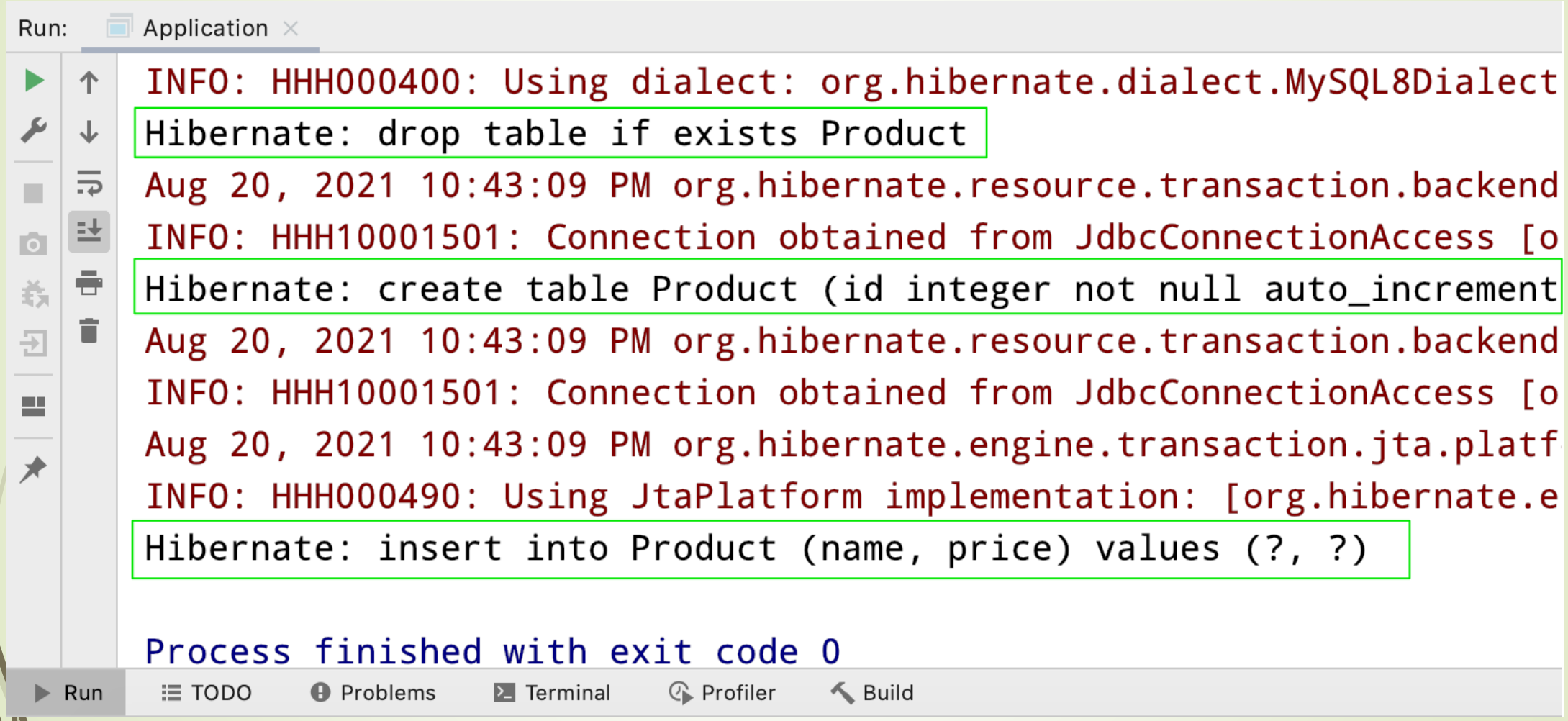
root

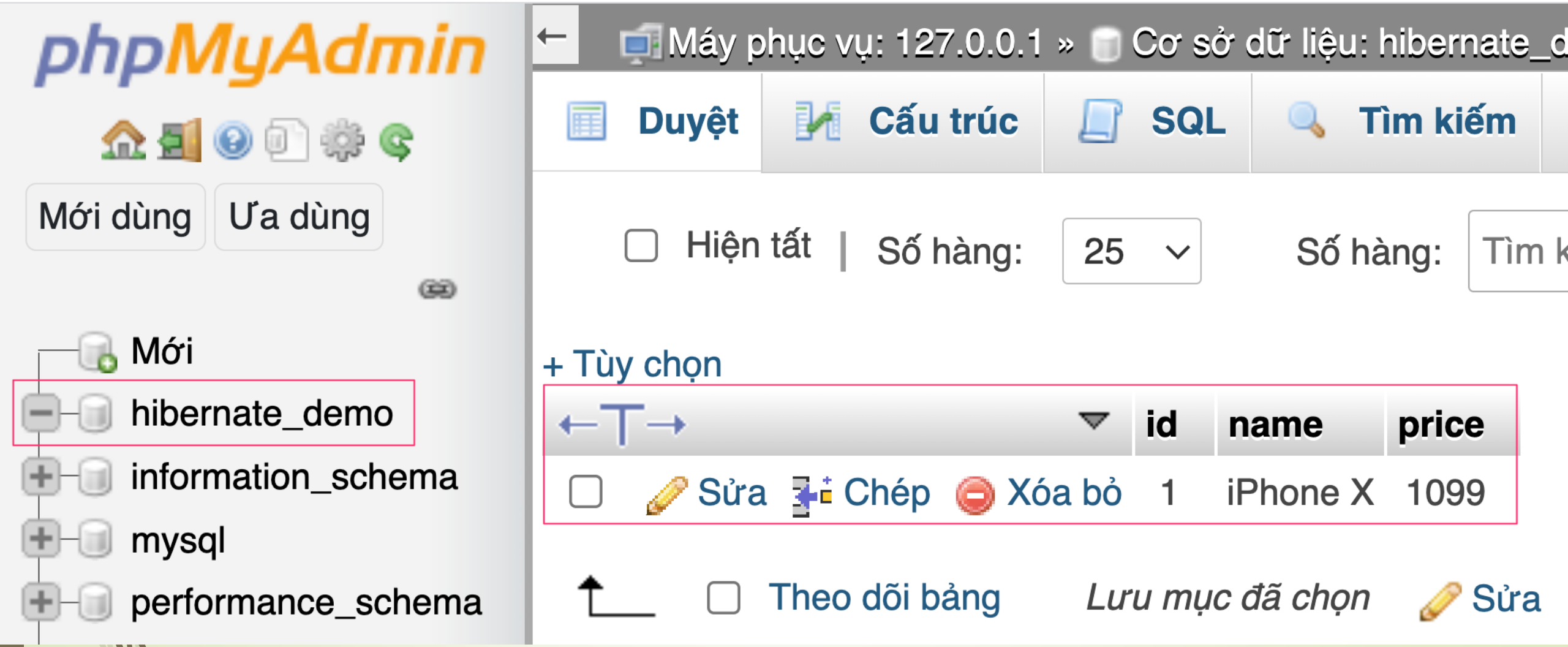
(empty)

main/java/Product.java

```java
@Entity
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;
    private double price;

    // default + parameterized constructors
    // getters + setters
    // toString
```

```
public class Application {
    public static void main(String[] args) {
        SessionFactory factory =  new Configuration()
            .configure()
            .addAnnotatedClass(Product.class)
            .buildSessionFactory();

        Session session = factory.openSession();

        Transaction transaction = session.beginTransaction();

        session.save(new Product( id: 1,  name: "iPhone X",  price: 1099));

        transaction.commit();

        session.close();

    }
}
```


Run: Application
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQL8Dialect
Hibernate: drop table if exists Product
Aug 20, 2021 10:43:09 PM org.hibernate.resource.transaction.backend
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [o
Hibernate: create table Product (id integer not null auto_increment
Aug 20, 2021 10:43:09 PM org.hibernate.resource.transaction.backend
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [o
Aug 20, 2021 10:43:09 PM org.hibernate.engine.transaction.jta.platf
INFO: HHH000490: Using JtaPlatform implementation: [org.hibernate.e
Hibernate: insert into Product (name, price) values (?, ?)
Process finished with exit code 0
Run
TODO
Problems
Terminal
Profiler
Build


phpMyAdmin
Mới dùng
Ưa dùng
Mới
hibernate_demo
information_schema
mysql
performance_schema
Máy phục vụ: 127.0.0.1 » Cơ sở dữ liệu: hibernate_d
Duyệt
Cấu trúc
SQL
Tìm kiếm
Hiện tất
Số hàng: 25
Số hàng:
Tìm k
+ Tùy chọn
id
name
price
Sửa
Chép
Xóa bỏ
1
iPhone X
1099
Theo dõi bảng
Lưu mục đã chọn
Sửa

Hibernate Query Language

Hibernate Query Language

- HQL is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties.
- HQL queries are translated by Hibernate into conventional SQL queries, which in turns perform action on database.
- You can use SQL statements directly with Hibernate using Native SQL, but I would recommend to use HQL.

HQL - FROM Clause

- You will use FROM clause if you want to load complete persistent objects into memory.
- Following is the simple syntax of using FROM clause:

```
String hql = "FROM Employee";
Query query = session.createQuery(hql);
List results = query.list();
```

HQL - AS Clause

- The AS clause can be used to assign aliases to the classes in your HQL queries, especially when you have the long queries.
- For instance, our previous simple example would be the following:

```
String hql = "FROM Employee as E";
Query query = session.createQuery(hql);
List results = query.list();
```

HQL - Select Clause

- The SELECT clause provides more control over the result set then the from clause.
- If you want to obtain few properties of objects instead of the complete object, use the SELECT clause

```
String hql = "SELECT E.email FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```

HQL - Where Clause

- If you want to narrow the specific objects that are returned from storage, you use the WHERE clause.
- Following is the simple syntax of using WHERE clause:

```
String hql = "FROM Employee E WHERE E.id = 10";
Query query = session.createQuery(hql);
List results = query.list();
```

HQL - OrderBy Clause

- To sort your HQL query's results, you will need to use the ORDER BY clause.
- You can order the results by any property on the objects in the result set either ascending (ASC) or descending (DESC).

```
String hql = "FROM Employee E WHERE E.id > 10
                ORDER BY E.salary DESC";
Query query = session.createQuery(hql);
List results = query.list();
```

HQL - GroupBy Clause

- This clause lets Hibernate pull information from the database and group it based on a value of an attribute and, typically, use the result to include an aggregate value.
- Following is the simple syntax of using GROUP BY clause.

```
String hql = "SELECT SUM(E.salary), E.firtName FROM Employee E " +
      "GROUP BY E.firstName";
Query query = session.createQuery(hql);
List results = query.list();
```

HQL – Named Parameters

- Hibernate supports named parameters in its HQL queries.
- This makes writing HQL queries that accept input from the user easy and you do not have to defend against SQL injection attacks.

```
String hql = "FROM Employee E WHERE E.id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("employee_id",10);
List results = query.list();
```

CRUD Operations

The Create Operation

- The required steps to create an entity:
  1. Create a new instance of the entity object.
  2. Get a new Session from the SesionFactory object
  3. Open a transaction
  4. Call session.save() method and pass the entity object as parameter.
  5. Commit the transaction
  6. Close the sesision

```
public class Application {
    public static void main(String[] args) {
        SessionFactory factory =  new Configuration()
            .configure()
            .addAnnotatedClass(Product.class)
            .buildSessionFactory();

        Session session = factory.openSession();

        Transaction transaction = session.beginTransaction();

        session.save(new Product( id: 1,  name: "iPhone X",  price: 1099));

        transaction.commit();

        session.close();

    }
}
```

The Read Operation

- The required steps to read an object by its id
  1. Get a new Session from the SesionFactory object
  2. Call the session.get() method, e.g. `Product p = session.get(Product.class, id)`
  3. Process the object as required.
  4. Close the session

```
public class Application {
    public static void main(String[] args) {

        SessionFactory factory =  new Configuration()
            .configure()
            .addAnnotatedClass(Product.class)
            .buildSessionFactory();

        Session session = factory.openSession();

        Product p = session.get(Product.class,  serializable: 2);
        System.out.println(p);

        session.close();


    }
}
```

The Read Operation

- The required steps to query a list of objects:
  1. Get a new Session from the SesionFactory object
  2. Call the session.createQuery() method and pass the HQL query as parameter
  3. Call the Query.getResultList() to get the result
  4. Process the returned list.

```
public class Application {
    public static void main(String[] args) {

        SessionFactory factory =  new Configuration()
            .configure()
            .addAnnotatedClass(Product.class)
            .buildSessionFactory();

        Session session = factory.openSession();
        Query query = session.createQuery( s: "FROM Product");
        List<Product> products = query.getResultList();
        for (Product p: products) {
            System.out.println(p);
        }
        session.close();
    }
}
```

The Update Operation

- The required steps to update an object
  1. Get a new Session from the SesionFactory object
  2. Begin a transaction
  3. Load the object from the database
  4. Update the object properties
  5. Call session.save() method to mask the object as being updated entity.
  6. Commit the transaction
  7. Close the session

```
public class Application {
    public static void main(String[] args) {

        // code tạo session factory ở đây

        Session session = factory.openSession();

        Transaction transaction = session.beginTransaction();

        Product product = session.get(Product.class, serializable: 1);

        product.setName("iPhone 10 Like new 99%");
        product.setPrice(499);

        session.save(product);

        transaction.commit();

        session.close();

    }
}
```

The Delete Operation

- The required steps to delete an object
  1. Get a new Session from the SesionFactory object
  2. Begin a transaction
  3. Load the object from the database
  4. Call session.delete() method to mask the object as deleted.
  5. Commit the transaction
  6. Close the session

```
public class Application {
    public static void main(String[] args) {

        // code tạo session factory ở đây

        Session session = factory.openSession();

        Transaction transaction = session.beginTransaction();

        Product product = session.get(Product.class, serializable: 1);

        session.delete(product);

        transaction.commit();

        session.close();


    }
}
```

Entity Relationship

- One to One Relationship
- Many to One Relationship
- Many to Many Relationship

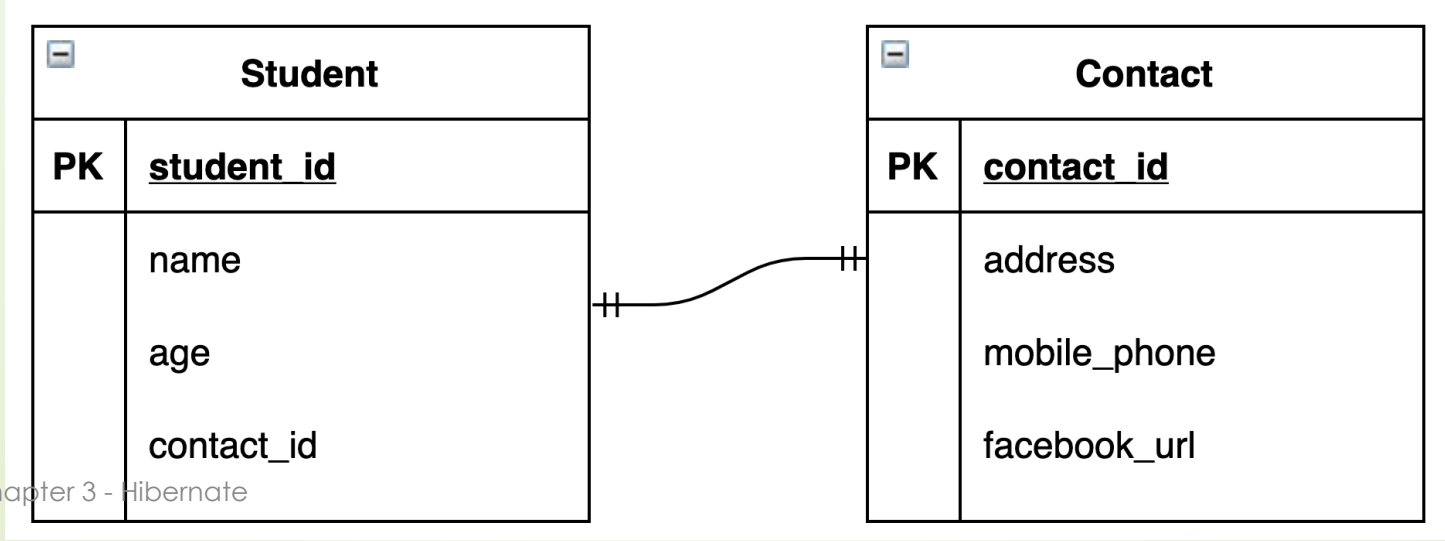
One to Many Relationship

One to One Relationship

- We have to use **@OneToOne** annotation to denote a one-to-one relationship in the entity classes.
- In hibernate there are 3 ways to create one-to-one relationships between two entities:
  1. Uses a foreign key column in one of the table.
  2. Use a third table to store mapping between first two tables.
  3. Uses a common primary key value in both the tables.

One to One Relationship

- **1**. Uses a foreign key column in one of the table:
  - In this kind of association, a foreign key column is created in owner entity.
  - Consider a relationship between Student and Contact:
    - An extra column contact_id will be created in the student table.
    - This column will store the foreign key for the Contact table.



One to One Relationship

- To make such association, refer the Contact entity in Student class as follow:

```
@Entity
public class Student {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;
    private int age;

    @JoinColumn(name = "contact_fk_id", referencedColumnName = "contact_id")
    @OneToOne
    private Contact contact;

    // getters, setters & toString
}
```

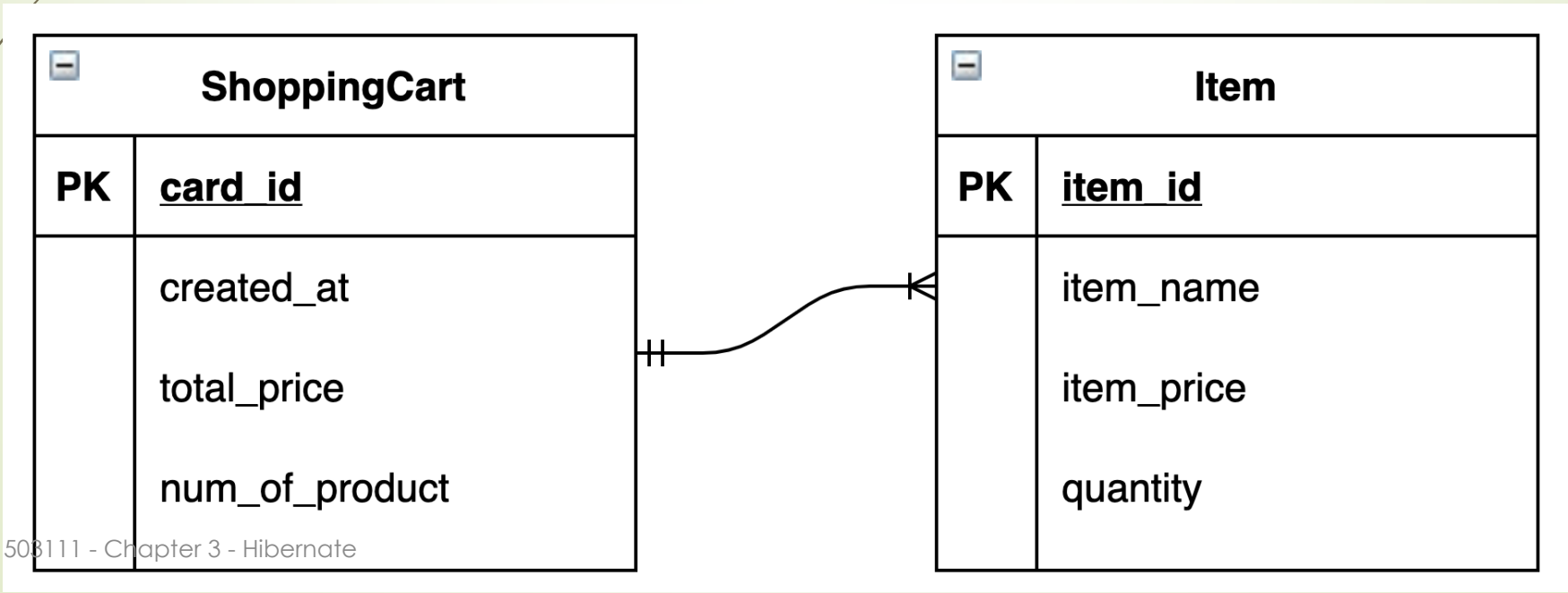
- In this seup the Student entity is called the owner. The owner is responsible for the association column(s) update.
- The other side is also annotated with @OneToOne annotation and the attribute mappedBy need to be specified.

```
@Entity
public class Contact {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "contact_id")
    private int id;

    private String adress;
    private String mobile_phone;
    private String facebook_url;

    @OneToOne(mappedBy = "contact")
    private Student student;
}
```

One to Many Relationship

One to Many Relationship

- One-to-many mapping means that one row in a table is mapped to multiple rows in another table.
- In this situation, we use @OneToMany annotation on ShoppingCart side and @ManyToOne on Item side.



ShoppingCart.java

```
@Entity
public class ShoppingCart {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private LocalDateTime createdAt;
    private int numOfProducts;
    private double totalPrice;


    @OneToMany(mappedBy = "cart", fetch = FetchType.EAGER)
    private List<Item> items;
}
```

Item.java

```
@Entity
public class Item {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String name;
    private double price;
    private int quantity;

    @JoinColumn(name = "cart_id", nullable = false)
    @ManyToOne
    private ShoppingCart cart;
}
```

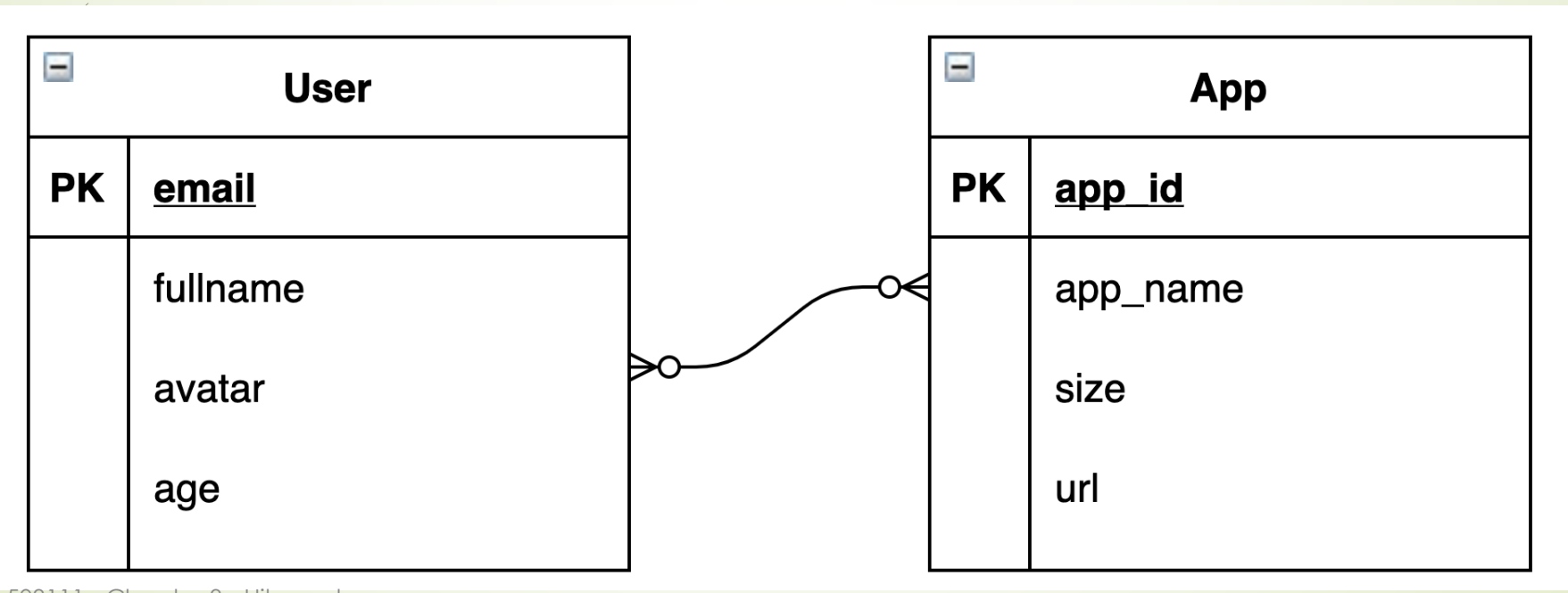
id	createdAt	numOfProducts	totalPrice
1	2021-08-21 16:56:09.085000	12	1099
2	2021-08-21 16:56:09.085000	15	1399

id	name	price	quantity	cart_id
1	Bàn chải	15	2	1
2	Dầu gội	25	1	1
3	Kem đánh răng	29	1	2
4	Mì gói	10	30	2
5	Sữa	15	15	2

Many to Many Relationship

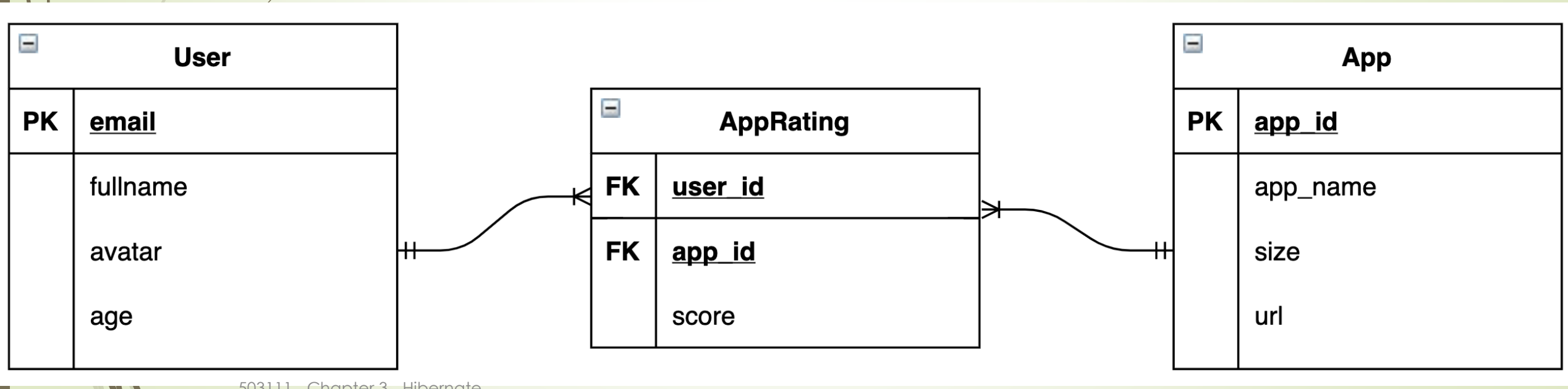
Many to Many Relationship

- In the case of a many-to-many relationship, both sides can relate to multiple instances of the other side.
- Let's take the example of user rating the app they like on App Store.



Many to Many Relationship

- Since both sides should be able to reference the other, we need to create a separate table to hold the foreign keys.
- In some cases, the many to many relationship also have properties.



```java
@Entity
public class AppRating {

    @EmbeddedId
    private AppRatingKey id;

    @JoinColumn(name = "user_id")
    @MapsId("userId")
    @ManyToOne
    private User user;

    @JoinColumn(name = "app_id")
    @MapsId("appId")
    @ManyToOne
    private App app;

    private double score;
}
```

```java
@Embeddable
public class AppRatingKey implements Serializable {

    @Column(name = "user_id")
    private int userId;

    @Column(name = "app_id")
    private int appId;

}
```

```java
@Entity
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String name;
    private String avatar;
    private int age;

    @OneToMany(mappedBy = "user", fetch = FetchType.EAGER)
    private List<AppRating> ratings = new ArrayList<>();

}
```

```java
@Entity
public class App {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String name;
    private double size;
    private String url;

    @OneToMany(mappedBy = "app", fetch = FetchType.EAGER)
    private List<AppRating> ratings = new ArrayList<>();

}
```

```java
@Entity
public class AppRating {

    /*private AppRatingKey id;
    private User user;
    private App app;
    private double score;*/

    public AppRating(User user, App app, double score) {
        this.user = user;
        this.app = app;
        this.score = score;
        this.id = new AppRatingKey(user.getId(), app.getId());
    }
}
```

	Bảng	Hành động
☐	App	Duyệt Cấu trúc
☐	AppRating	Duyệt Cấu trúc
☐	User	Duyệt Cấu trúc
	3 bảng	Tổng

id	name	size	url
1	Zalo	50	http://zalo.vn
2	Tiktok	75	http://tiktok.com

id	age	avatar	name
1	29	cat.jpg	Tam
2	30	pig.jpg	Dung
3	27	dog.jpg	Duy
4	22	bird.jpg	Pham
5	28	flower.jpg	Thi

app_id	user_id	score
1	1	6
1	2	5.5
1	3	8.3
1	4	5.2
2	1	8
2	2	7.5
2	4	8.7