Hibernate

132. Hibernate Section Introduction

Why Hibernate even exists (the pain first)

Before Hibernate, Java developers used **JDBC directly**.
Let's visualize old-school JDBC:

```
Connection con = DriverManager×getConnection(×××);
PreparedStatement ps = con×prepareStatement(
  "INSERT INTO user VALUES (?, ?)"
);
ps.setInt(1, 1);
ps.setString(2, "Sagar");
ps.executeUpdate();
```

Looks fine… until:

- 20 tables
- 100 queries
- Mapping rows to objects manually
- Handling transactions
- Handling connection leaks
- Database change breaks half the code

JDBC is powerful but very low-level.
Hibernate enters like an **automatic gearbox** 🚗
You still drive, but it handles the clutch.

Hibernate's promise

Hibernate says:
"Stop thinking in tables. Think in **objects**."
Instead of:

- Rows
- Columns
- Foreign keys

You work with:

- Classes
- Objects
- Relationships

Hibernate silently translates **Java objects ↔ SQL queries**.
This entire idea is called:
👉 **ORM – Object Relational Mapping**

133. What is Hibernate

Definition (interview-ready)

Hibernate is a **Java ORM framework** that maps Java objects to relational database tables and manages database operations automatically.
Shorter version for interviews:
Hibernate lets you persist Java objects into a database without writing SQL.

Real-world analogy

Imagine a **translator** sitting between you and the database.

- You speak **Java**
- Database speaks **SQL**
- Hibernate is the translator

You say:

```
session.save(user);
```

Hibernate hears:

```
INSERT INTO user_table (id, name) VALUES (?, ?);
```

You never typed SQL. Hibernate did.

JDBC vs Hibernate (crucial for interviews)

JDBC	Hibernate
Manual SQL	Auto-generated SQL
Tight DB coupling	DB independent
Boilerplate code	Clean object code
Error-prone	Safer & structured
Faster for small apps	Scales better

Important truth (senior insight):
Hibernate **uses JDBC internally**. It doesn't replace it, it **abstracts it**.

Core Hibernate idea (burn this into memory)

Hibernate is based on **three pillars**:

1. **Entity** – Java class mapped to a table
2. **Session** – Interface to talk to DB
3. **Transaction** – Ensures ACID safety

We'll explore each deeply later.

Tiny sneak peek example

Java class:

```
@Entity
public class User {
  @Id
  private int id;
```

```
  private String name;
}
```

Saving data:

```
session.save(user);
```

That's it.
No SQL.
No ResultSet.
No PreparedStatement.
Hibernate handles:

- SQL
- Connections
- Mapping
- Transactions

Common interview trap ❌

Question: Is Hibernate a framework or API?
Correct answer:
Hibernate is a **framework** that provides ORM, caching, HQL, and transaction management.

Big Picture: What do we need for Hibernate?

A Hibernate project needs **5 essential things**:

1. **Java Project**
2. **Hibernate Libraries**
3. **Database Driver**
4. **Configuration file**
5. **Entity class**

Think of it like setting up a new office:

- Room (Java project)
- Staff (Hibernate jars)
- Telephone line (DB driver)
- Rules book (config)
- Employees (entities)

Step 1: Project Structure (simple mental model)

```
hibernate-demo
 ├── src/main/java
 │    └── com.example
 │         └── App.java
 │         └── User.java   ← Entity
 ├── src/main/resources
```

```
│       └── hibernate.cfg.xml
└── pom.xml
```

We'll assume **Maven**, because interviews expect that.

Step 2: pom.xml (Hibernate + DB)

Core dependencies:

```
<dependencies>

    <!-- Hibernate Core -->
    <dependency>
        <groupId>org.hibernate.orm</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>6.6.36.Final</version>
    </dependency>


    <!-- https://mvnrepository.com/artifact/org.postgresql/
postgresql -->
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>42.7.3</version>
    </dependency>

</dependencies>
```

🔷 Hibernate doesn't connect to DB alone
🔷 JDBC driver is still required
Interview line:
Hibernate is DB-independent, but JDBC drivers are DB-specific.

Step 3: hibernate.cfg.xml (brain of Hibernate)

```
<hibernate-configuration>
    <session-factory>

        <property name="hibernate.connection.driver_class">
            org.postgresql.Driver
        </property>

        <!-- Default PostgreSQL port is 5432 -->
        <property name="hibernate.connection.url">
            jdbc:postgresql://localhost:5432/testdb
        </property>
```

```
        <!-- Use your PostgreSQL username and password -->
        <property
name="hibernate.connection.username">postgres</property>
        <property
name="hibernate.connection.password">root</property>

        <!-- Use the appropriate dialect for PostgreSQL -->
        <property name="hibernate.dialect">
            org.hibernate.dialect.PostgreSQLDialect
        </property>

        <property name="hibernate.hbm2ddl.auto">update</
property>

        <property name="show_sql">true</property>

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

🔥 Important interview point:

- dialect tells Hibernate **which SQL to generate**
- That's how DB independence is achieved

Step 4: Entity class

```
@Entity
public class User {

    @Id
    private int id;

    private String name;

    // getter and setters for the fields
}
```

This class:

- Represents a table
- Each object represents a row

Hibernate magic starts here ✨

Step 5: Bootstrapping Hibernate

```
Configuration cfg = new Configuration();
cfg.configure();
```

```
SessionFactory factory = cfg×buildSessionFactory();
Session session = factory×openSession();
```

Creat configuration, build session factory and then open a session
Create object, save object, close the session

Mental model:

- **SessionFactory** → Heavy, created once
- **Session** → Lightweight, per request

Before moving ahead, one check 🧩

Answer **only this**:
Q:
Why is hibernate.dialect required even though JDBC driver is already provided?

1️⃣ JDBC Driver answers:

How do I connect to the database?
Example:

- How to open socket
- How to send SQL text
- How to receive result set

That's all.

2️⃣ Hibernate Dialect answers:

What kind of SQL should I generate?
Example:

- LIMIT vs ROWNUM
- AUTO_INCREMENT vs SEQUENCE
- BOOLEAN vs TINYINT
- Pagination syntax
- Date/time functions

Same Java code:

```
session.createQuery("from User").list();
```

Generated SQL differs:

- MySQL ➜ LIMIT
- Oracle ➜ ROWNUM
- PostgreSQL ➜ LIMIT OFFSET

🧠 **Dialect = SQL grammar book**

Tiny memory hook 🪝

- **Driver** → *How to talk*
- **Dialect** → *What language to speak*

135. Failed Attempt to Save Data

This topic explains:

- Why beginners say “Hibernate is not saving data”
- Why no error comes but no row is inserted
- Why transaction matters more than save()

Scenario: Beginner writes this code

```
Session session = factory×openSession();

User user = new User();
user.setId(1);
user.setName("Sagar");

session.save(user);

session.close();
```

This code:

- Compiles
- Runs
- Shows **no error**

Yet… **data may NOT be saved** 😈
This is the classic beginner trap.

Why this code *looks correct* but is actually incomplete

Let’s pause and reason.

Hibernate follows this rule:

No transaction = no guarantee of database write
Hibernate does **not auto-commit** like JDBC sometimes does.
So what happens here:

- save() only puts object in **Hibernate memory**
- No transaction
- No commit
- Hibernate silently discards changes

This leads us directly to: Failed Attempt to Save Data (core concept)

The missing piece ❌ 👉 Transaction

Correct App.java (version 2 – working)

```
package com.example;

import org.hibernate.Session;
```

```
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

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

        Configuration cfg = new Configuration();
        cfg.configure();

        SessionFactory factory = cfg×buildSessionFactory();
        Session session = factory×openSession();

        // 🔑 Start transaction
        Transaction tx = session×beginTransaction();

        User user = new User();
        user.setId(1);
        user.setName("Sagar");

        session.save(user);

        // 🔑 Commit transaction
        tx.commit();

        session.close();
        factory.close();
    }
}
```

Now:

- SQL is executed
- Row is inserted
- Data is **actually persisted**

Interview gold insight 🪙

If interviewer asks:

Q: Why is transaction mandatory in Hibernate?

Answer:

Without commit, Hibernate doesn't flush changes to DB.

Mental model to remember 🧠

- save() → marks object for persistence
- beginTransaction() → starts unit of work
- commit() → actually hits the database

No commit = no insert.

Quick check (answer in 1 line)

Q:
What exactly happens if session.save() is called without a transaction?
session.save() only moves the object into Hibernate's persistence context. Without a transaction commit, Hibernate does not flush the changes to the database, so the data is discarded when the session closes.

136. Successful Attempt to Save Data

This topic is not about "it works"
It's about **why** it works.

The full correct flow (burn this in memory 🔥)

```
Session session = factory×openSession();
Transaction tx = session×beginTransaction();

session.save(user);

tx.commit();
session.close();
```

What happens internally (step-by-step):

1. openSession()
   → Creates **persistence context** (1st level cache)
2. beginTransaction()
   → Starts a **unit of work**
3. save(user)
   → Object becomes **persistent**
   → Stored in Hibernate memory
4. commit()
   → Hibernate **flushes SQL**
   → JDBC executes INSERT
   → DB row is created
5. close()
   → Persistence context destroyed

Important interview keyword: Flush

Hibernate does **not execute SQL immediately**.
It executes SQL when:

- Transaction commits
- Session flush happens

137. show_sql configuration (seeing is believing)

Add this to hibernate.cfg.xml:

```
<property name="hibernate.show_sql">true</property>
<property name="hibernate.format_sql">true</property>
```

Now when you run App.java, you'll see:

```
Hibernate:
    insert
    into
        User
        (name, id)
    values
        (?, ?)
```

🧠 This is how you *prove* Hibernate is generating SQL.

Interview pitfall ⚠

Q: Does show_sql affect performance?
A:
Yes. It should be enabled only for development/debugging, not production.

138. Refactoring (Senior Dev Thinking)

Right now, every app does this:

```
Configuration cfg = new Configuration();
cfg.configure();
SessionFactory factory = cfg.buildSessionFactory();
```

Creating SessionFactory is **heavy**.
We should do it **once**.

Refactor into HibernateUtil

```
public class HibernateUtil {

    private static SessionFactory factory;

    static {
        try {
            factory = new
Configuration().configure().buildSessionFactory();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    public static SessionFactory getSessionFactory() {
        return factory;
```

```
    }
}
```

Now App.java becomes clean:

```
Session session =
HibernateUtil×getSessionFactory()×openSession();
Transaction tx = session×beginTransaction();

session.save(user);

tx.commit();
session.close();
```

Interview keyword:
SessionFactory is thread-safe; Session is not.

Quick checkpoint 🧩

Answer this:
Q:
Why is SessionFactory created once but Session created multiple times?
SessionFactory is a heavy, thread-safe object that holds configuration and metadata, so it is created once. Session is lightweight and not thread-safe, so it is created per request or unit of work.

139. Fetching the Data

Saving is only half the story. Fetching is where Hibernate feels magical ✨

Two primary ways to fetch data

Hibernate gives you **two fundamental methods**:

1. get()
2. load()

(We'll deep-dive later, but first basics.)

Fetch by Primary Key (most common)

```
Session session =
HibernateUtil×getSessionFactory()×openSession();

User user = session×get(User×class, 1);

System.out.println(user.getName());

session.close();
```

What happens internally:

- Hibernate checks **1st level cache**
- If not found, hits DB
- Executes SELECT

SQL you'll see (with show_sql)

```
select
    user0_.id as id1_0_0_,
    user0_.name as name2_0_0_
from
    User user0_
where
    user0_×id=?
```

What if record doesn't exist?

```
User user = session.get(User.class, 99);
```

Result:

```
user == null
```

No exception. Clean behavior.

Fetching multiple records

```
List<User> users = session
        .createQuery("from User", User.class)
        .list();
```

- from User → HQL (Hibernate Query Language)
- Returns list of objects, not rows

140. Update and Delete Objects

Hibernate works on **object state**, not SQL.

Updating an object

```
Session session =
HibernateUtil×getSessionFactory()×openSession();
Transaction tx = session×beginTransaction();

User user = session×get(User×class, 1);
user.setName("Updated Name");
```

```
tx.commit();
session.close();
```

🧠 Key insight:

- No update() call required
- Hibernate detects change automatically (**dirty checking**)

Deleting an object

```
Session session =
HibernateUtil×getSessionFactory()×openSession();
Transaction tx = session×beginTransaction();

User user = session.get(User.class, 1);
session.delete(user);

tx.commit();
session.close();
```

Interview gold: Object States 🪙

Hibernate tracks objects in states:

1. **Transient** – new object, not saved
2. **Persistent** – attached to session
3. **Detached** – session closed
4. **Removed** – marked for deletion

Understanding this explains **90% Hibernate behavior**.

Quick check (single question)

Q:

Why does Hibernate not need an explicit update() call when modifying a fetched object?

Hibernate performs **dirty checking**, so it automatically detects changes in a persistent object and generates the required UPDATE SQL at transaction commit.

141. Changing Table and Column Names

By default, Hibernate follows simple rules:

- Class name → table name
- Field name → column name

```
public class User {
    private int id;
    private String name;
}
```

Hibernate assumes:

```
Table: User
Columns: id, name
```

But real databases rarely look this clean.

Custom table name

```
@Entity
@Table(name = "user_details")
public class User {
    @Id
    private int id;

    private String name;
}
```

Now Hibernate maps:

- Class User → table user_details

Custom column names

```
@Entity
@Table(name = "user_details")
public class User {

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

    @Column(name = "user_name")
    private String name;
}
```

🧠 Hibernate does **object → column mapping**, not magic guessing.

Interview insight 🪨

Annotations decouple Java naming conventions from database naming conventions.
This is very commonly asked.

142. Embeddable

This is about **grouping fields**, not tables.

Problem first

```
public class User {
    private String street;
    private String city;
    private String pincode;
}
```

This clutters the entity.

Solution: @Embeddable
Step 1: Create value object

```
@Embeddable
public class Address {
    private String street;
    private String city;
    private String pincode;
}
```

Step 2: Embed it

```
@Entity
public class User {

    @Id
    private int id;

    private String name;

    @Embedded
    private Address address;
}
```

What happens in DB?
Still **ONE table**:

```
User
_____
id
name
street
city
pincode
```

No separate address table.

Important distinction (interview favorite)

Embeddable	Entity
No identity	Has primary key
No table	Own table
Value object	Domain object
Part of parent	Independent

Mental model 🧠

- @Entity → **has life of its own**
- @Embeddable → **dies with parent**

Quick checkpoint (one line)

Q:
When would you choose @Embeddable instead of a separate entity?
We use @Embeddable when a group of fields logically belongs together and does not have its own identity or lifecycle, and we want to avoid cluttering the entity while keeping everything in the same table.

143. Mapping Relationship Theory

Before writing annotations, we must understand **how relationships actually work**.

Fundamental truth (burn this in memory 🔥)

👉 **Relationships exist in the database, not in Java.**
Java just *models* them.
Database relationships are based on:

- Primary Key
- Foreign Key

Hibernate just maps that reality.

Types of relationships (theory first)

From DB perspective, only these exist:

1. One-to-One
2. One-to-Many
3. Many-to-One
4. Many-to-Many

Hibernate mirrors them using annotations.

Ownership concept (MOST IMPORTANT)

Every relationship has:

- **Owning side**
- **Inverse side**

Rule:

👉 **The side with the foreign key is the owning side**
Hibernate only listens to the owning side when generating SQL.

Example: User and Address
Database:

```
user
_____
id (PK)
name
address_id (FK)
```

Who owns the relationship?
- User (because it has FK)

Common beginner mistake ❌
Updating the **inverse side** and expecting DB changes.
Hibernate ignores it.

mappedBy (interview killer keyword)
mappedBy tells Hibernate:
"I am NOT the owner. Look at the other side."

Direction matters

Mapping	Meaning
Unidirectional	One side knows
Bidirectional	Both sides know

DB doesn't care. Java does.

Quick mental checklist before mapping
Ask yourself:
1. Where is the foreign key?
2. Who owns the relationship?
3. Do I need navigation from both sides?

If you answer these, annotations become easy.

Checkpoint (very important)
Answer this carefully:
Q:
In a bidirectional relationship, which side controls the database update and why?
In a bidirectional relationship, the owning side controls database updates. The owning side is the table that contains the foreign key column.

144. One-to-One Mapping
We'll start with the **simplest and safest** form.

Scenario

One **User** has **one Passport**
One **Passport** belongs to **one User**

Database design

```
user
----
id (PK)
name

passport
--------
id (PK)
number
user_id (FK)
```

Foreign key is in passport
So **Passport is the owning side**

Step 1: User entity (inverse side)

```
@Entity
public class User {

    @Id
    private int id;

    private String name;

    @OneToOne(mappedBy = "user")
    private Passport passport;
}
```

- mappedBy = "user" → User is **not owner**
- Hibernate will ignore this side for SQL updates

Step 2: Passport entity (owning side)

```
@Entity
public class Passport {

    @Id
    private int id;

    private String number;

    @OneToOne
```

```
    @JoinColumn(name = "user_id")
    private User user;
}
```

- @JoinColumn → creates foreign key
- This side controls DB

Saving data (IMPORTANT order)

```
User user = new User();
user.setId(1);
user.setName("Sagar");

Passport passport = new Passport();
passport.setId(101);
passport.setNumber("IND12345");

passport.setUser(user);
user.setPassport(passport);

session.save(user);
session.save(passport);
```

🧠 Always set **both sides in Java**
Hibernate won't auto-sync object references.

Interview insight 🪨
mappedBy prevents duplicate foreign keys and tells Hibernate which side owns the relationship.

Common mistake ❌
Putting @JoinColumn on **both** entities
→ Results in extra column or mapping error

Quick check
Q:
What happens if mappedBy is removed from the inverse side?
If mappedBy is removed, Hibernate treats both sides as owning sides and creates an additional foreign key or join column, leading to redundant columns or mapping errors.
Key idea:
Hibernate is not "confused" — it assumes **two separate relationships**.

145. OneToMany and ManyToOne
This shows up **everywhere**: orders, employees, trades, transactions.

Scenario

One **Department** has many **Employees**
Each **Employee** belongs to one **Department**

Database design

```
department
----------
id (PK)
name

employee
--------
id (PK)
name
department_id (FK)
```

Foreign key is in employee
So **Employee is the owning side**

Step 1: ManyToOne (OWNING SIDE)

```
@Entity
public class Employee {

    @Id
    private int id;

    private String name;

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;
}
```

This side:

- Has FK
- Controls DB updates

Step 2: OneToMany (INVERSE SIDE)

```
@Entity
public class Department {

    @Id
    private int id;
```

```
    private String name;

    @OneToMany(mappedBy = "department")
    private List<Employee> employees;
}
```

- mappedBy points to field in Employee
- No join column here

Saving data correctly

```
Department dept = new Department();
dept.setId(1);
dept.setName("IT");

Employee e1 = new Employee();
e1.setId(101);
e1.setName("A");

Employee e2 = new Employee();
e2.setId(102);
e2.setName("B");

e1.setDepartment(dept);
e2.setDepartment(dept);

session.save(dept);
session.save(e1);
session.save(e2);
```

🧠 Always set the **owning side** (Employee).

Interview gold 🪙

Q: Why is ManyToOne preferred over OneToMany?
A:
Because ManyToOne avoids join tables and maps naturally to foreign keys.

Common mistake ❌

Putting @JoinColumn on @OneToMany
→ Hibernate creates an unnecessary join table

Quick checkpoint

Q:
Why does Hibernate create a join table if mappedBy is missing in OneToMany?
Hibernate creates a join table when mappedBy is missing because it treats the

relationship as **two independent owning sides** and cannot place a foreign key in either table.

Hibernate's logic is simple:

- No owner defined
- No clear foreign key location
- So it creates a **join table** to manage the association

That's it.

146. Many-to-Many Mapping

This is the **most misunderstood** mapping. We'll keep it clean.

Scenario

A **Student** can enroll in many **Courses**
A **Course** can have many **Students**

Database reality

You **cannot** store this with a single foreign key.
So DB uses a **join table**:

```
student
_______
id
name

course
______
id
title

student_course
______________
student_id (FK)
course_id (FK)
```

Join table is **mandatory** in many-to-many.

Step 1: Owning side

```
@Entity
public class Student {

    @Id
    private int id;
```

```
    private String name;

    @ManyToMany
    @JoinTable(
        name = "student_course",
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "course_id")
    )
    private List<Course> courses;
}
```

This side:

- Defines join table
- Owns the relationship

Step 2: Inverse side

```
@Entity
public class Course {

    @Id
    private int id;

    private String title;

    @ManyToMany(mappedBy = "courses")
    private List<Student> students;
}
```

- mappedBy points to owning field
- No join table here

Saving data (important)

```
Student s = new Student();
s.setId(1);
s.setName("Sagar");

Course c1 = new Course();
c1.setId(101);
c1.setTitle("Hibernate");

s.getCourses().add(c1);
c1.getStudents().add(s);
```

```
session.save(c1);
session.save(s);
```

🧠 Always update **both sides in Java**.

Interview insight 🪨

Q: Why is ManyToMany discouraged in real systems?
A:
Because join tables often need extra columns (date, status), which requires converting it into two OneToMany relationships.

When do we replace ManyToMany with an intermediate entity?

When the join table needs **additional attributes** or **business meaning**.
That's it. That's the rule.

Why ManyToMany breaks in real systems

Pure ManyToMany assumes:

```
student_id | course_id
```

But real life asks for more:

- enrollment_date
- status
- grade
- payment_status

Now the join table is no longer "just a join".
It becomes a **real entity**.

Correct redesign (interview gold 🪨)

Instead of:

```
Student ⇔ Course (ManyToMany)
```

We do:

```
Student → Enrollment ← Course
```

Example

```
@Entity
public class Enrollment {

    @Id
    private int id;

    @ManyToOne
    private Student student;
```

```
    @ManyToOne
    private Course course;

    private LocalDate enrolledDate;
    private String status;
}
```

Now:

- No ManyToMany
- Two ManyToOne
- Clean, extensible, production-ready

147. Hibernate Eager and Lazy Fetch

This topic causes:

- Performance issues
- N+1 problem
- Production outages 😄

So we'll go carefully.

Default fetching rules (VERY IMPORTANT)

Mapping	Default Fetch
ManyToOne	EAGER
OneToOne	EAGER
OneToMany	LAZY
ManyToMany	LAZY

Memorize this table.

What is LAZY?

Hibernate loads data **only when accessed**.

```
Department dept = session.get(Department.class, 1);

// No employee query yet

dept.getEmployees(); // SQL fires here
```

What is EAGER?

Hibernate loads data **immediately with parent**.

```
Department dept = session.get(Department.class, 1);
// Employees loaded immediately
```

Why LAZY is default for collections?

Because:

- Collections can be huge
- Loading them blindly kills performance

Common production bug ⚠

```
session.close();
dept.getEmployees(); // 💥 LazyInitializationException
```

Because:
- Session is closed
- Proxy can't fetch data

Interview insight 🪨
Q: How do you solve LazyInitializationException?
A:
- Open Session in View
- Fetch Join
- Initialize before session close

(We'll revisit in HQL.)

Quick check (one line)
Q:
Why is ManyToOne EAGER by default, but OneToMany LAZY?
ManyToOne → EAGER
Example:

```
Employee → Department
```

- Each **Employee** has **one** Department
- Fetching one extra row is cheap
- Almost always required in business logic

So Hibernate assumes:
"If you load an employee, you'll probably need its department."
Hence:

```
@ManyToOne   // default fetch = EAGER
private Department department;
```

OneToMany → LAZY
Example:

```
Department → Employees
```

- One Department can have **hundreds or thousands** of employees
- Fetching all employees blindly is expensive
- Often not required immediately

So Hibernate assumes:

“Load this only if the developer explicitly asks.”
Hence:

```
@OneToMany   // default fetch = LAZY
private List<Employee> employees;
```

The core principle (remember this)
Hibernate defaults are optimized for **performance safety**, not convenience.

Interview-ready one-liner 🎯
ManyToOne is EAGER because it fetches a single associated object, which is cheap and commonly needed, whereas OneToMany is LAZY because collections can be large and expensive to load.

148. Hibernate Caching
This is **high-impact interview content**.

What problem does caching solve?
Repeated DB hits.

```
User u1 = session.get(User.class, 1);
User u2 = session.get(User.class, 1);
```
Without cache:
- Two SELECTs

With cache:
- One SELECT

1️⃣ First Level Cache (Session Cache)
- Enabled by default
- Scope: **Session**
- Cannot be disabled

```
Session s = factory.openSession();
s.get(User.class, 1); // hits DB
s.get(User.class, 1); // from cache
```

No config needed.

2️⃣ Second Level Cache (SessionFactory Cache)
- Optional
- Shared across sessions
- Needs configuration (Ehcache, etc.)

```
Session s1 = factory.openSession();
Session s2 = factory.openSession();
```

```
s1.get(User.class, 1); // DB
s2.get(User.class, 1); // Cache
```

Interview insight 🪨
First-level cache is mandatory and per session.
Second-level cache is optional and per SessionFactory.

Quick checkpoint (one line)
Q:
Why can’t first-level cache be shared across sessions?
First-level cache is tied to a Session because a Session represents a single unit of work and is not thread-safe. Sharing it across sessions could lead to data inconsistency and concurrency issues.
Key ideas:

- Session = unit of work
- Not thread-safe
- Isolation matters

149. HQL Introduction
What is HQL?
HQL (Hibernate Query Language) is:

- Object-oriented
- Database independent
- Works on **entities**, not tables

```
from User
```

Not:

```
SELECT * FROM user_table
```

Why HQL?

- No table names
- No column names
- Hibernate converts it to SQL using Dialect

150. Fetching Data using HQL

```
List<User> users =
    session.createQuery("from User", User.class)
           .list();
```
Fetch by condition:

```
User u = session.createQuery(
    "from User where name = :name", User.class)
```

```
    .setParameter("name", "Sagar")
    .uniqueResult();
```

151. Fetching with filter and specific properties

Selecting specific fields (projection)

```
List<String> names =
    session.createQuery(
        "select name from User", String.class)
        .list();
```

Multiple fields

```
List<Object[]> data =
    session.createQuery(
        "select id, name from User", Object[].class)
        .list();
```

Interview tip:
Projections reduce memory usage.

152. Get vs Load (VERY IMPORTANT)

get()	load()
Hits DB immediately	Returns proxy
Returns null	Throws exception
Safe	Faster
Use when unsure	Use when sure

```
session.get(User.class, 1);
session.load(User.class, 1);
```

When load() fails

```
User u = session.load(User.class, 99);
u.getName(); // ObjectNotFoundException
```

153. Level 2 Cache using Ehcache

Why Level 2 cache?

- Share cache across sessions
- Reduce DB load

Steps (high level)

1. Add Ehcache dependency
2. Enable 2nd level cache
3. Mark entities cacheable

```
@Entity
@Cacheable
@org.hibernate.annotations.Cache(
    usage = CacheConcurrencyStrategy.READ_WRITE
)
public class User {
    ...
}
```

Interview insight 🪨

Q: Is caching suitable for all entities?
A:
No. Only for rarely-changing, frequently-read data.

Hibernate lifecycle

Hibernate lifecycle starts with bootstrapping, where a SessionFactory is created using configuration and mappings. SessionFactory is a heavy, thread-safe object created once for the application.
For each unit of work, Hibernate opens a Session, which represents a persistence context and provides first-level caching. A transaction is started to group database operations.
Hibernate entities move through different states. A new object is in a **transient** state. When it is associated with a session using save or persist, it becomes **persistent**, and Hibernate starts tracking it. Any changes made to a persistent object are automatically detected through dirty checking and synchronized with the database at transaction commit.
When the session is closed, the entity becomes **detached**. It is no longer tracked, and changes won't be persisted unless reattached. If an entity is deleted, it enters the **removed** state.
Hibernate executes SQL during flush or transaction commit and uses caching and fetch strategies like lazy and eager loading to optimize performance.