Hibernate:

**✅ Important Hibernate Advantages for Interview**

1. **Simplifies JDBC code** – No need to write complex SQL; Hibernate handles it.
2. **Object Relational Mapping (ORM)** – Maps Java objects to database tables.
3. **Automatic Table Creation** – Can generate tables based on entity classes.
4. **HQL (Hibernate Query Language)** – More object-oriented than SQL.
5. **Caching Support** – First-level and second-level cache for better performance.

**Note**:

**🧠 What is Caching in Hibernate?**

Caching helps reduce **unnecessary database hits** by storing frequently accessed data in memory.

Hibernate supports two types:

**✅ 1. First-Level Cache (Enabled by Default)**

* It's associated **with the Hibernate Session**.
* If you fetch the same entity twice **in the same session**, Hibernate gets it from the **cache, not the DB**.

**🔄 Example:**

java

CopyEdit

Session session = sessionFactory.openSession();

Employee emp1 = session.get(Employee.class, 1L); // Hits DB

Employee emp2 = session.get(Employee.class, 1L); // Comes from cache

🟢 Only one DB query is executed here.

**✅ 2. Second-Level Cache (Optional – Must Enable)**

* Works **across sessions**.
* Hibernate uses providers like **EhCache, Redis, or Infinispan** to store data globally.
* Once cached, entities can be reused in other sessions without hitting the DB.

**🧾 To Enable (Example with EhCache):**

properties

CopyEdit

spring.jpa.properties.hibernate.cache.use\_second\_level\_cache=true

spring.jpa.properties.hibernate.cache.region.factory\_class=org.hibernate.cache.ehcache.EhCacheRegionFactory

And annotate your entity:

java

CopyEdit

@Cacheable

@org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.READ\_WRITE)

@Entity

public class Employee {

...

}

**💬 Interview One-Liner:**

“Hibernate supports first-level caching by default at the session level, and optional second-level caching across sessions using providers like EhCache to reduce DB hits and boost performance.”

1. **Lazy Loading** – Loads data only when needed, improving efficiency.

**Note**:

**Lazy loading** means:

Hibernate **doesn’t load associated data from the database until you actually access it** in your code.

This saves memory and improves performance.

**✅ Example**

Let’s say you have two entities:

**🧾 Department and Employee**

@Entity

public class Department {

@Id

private Long id;

private String name;

@OneToMany(mappedBy = "department", fetch = FetchType.LAZY)

private List<Employee> employees;

}

Here:

* When you fetch a Department, the list of employees **is not fetched immediately**.
* Only when you call department.getEmployees(), Hibernate hits the database and loads them.

1. **Database Independence** – Easily switch between different databases.

**Note**:

**📘 What is a Dialect in Hibernate?**

A **Dialect** in Hibernate is a **class that tells Hibernate how to convert HQL (Hibernate Query Language) into SQL** that's specific to the database you are using.

Different databases (like MySQL, PostgreSQL, Oracle, etc.) have different SQL syntax, so Hibernate uses dialects to generate the correct SQL.

**🔄 How Hibernate Provides Database Independence**

Hibernate uses **dialects** to generate database-specific SQL behind the scenes. You just need to:

**✅ Steps to Switch Databases in Hibernate**

1. **Change the Dialect**  
   In your application.properties or hibernate.cfg.xml, update the dialect:

properties

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect

# Or switch to PostgreSQL

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect

1. **Update the JDBC URL, Username, Password**

properties

spring.datasource.url=jdbc:mysql://localhost:3306/yourdb

spring.datasource.username=root

spring.datasource.password=yourpassword

1. ✅ **That's it!** Hibernate handles the rest (table creation, query generation) using the new database dialect.
2. **Transaction Management** – Integrated support for managing transactions.

**Note**:

**✅ 1. Hibernate's Native API**

Using Transaction interface:

Session session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();

try {

// Perform DB operations

tx.commit();

} catch (Exception e) {

tx.rollback(); // Rollback if error

}

**✅ 2. Spring + Hibernate Transaction Management Example**

Assume you have an entity called Employee:

**🧾 Employee.java**

java

CopyEdit

@Entity

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String department;

}

**🧾 EmployeeRepository.java**

You use **Spring Data JPA** for DB operations:

java

CopyEdit

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

}

**🧾 EmployeeService.java**

This is where you manage transactions using @Transactional:

java

CopyEdit

@Service

public class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

@Transactional

public void addEmployee(Employee employee) {

employeeRepository.save(employee);

// Example logic that might throw an exception

if (employee.getName().equals("Error")) {

throw new RuntimeException("Simulated error");

}

}

}

“With Spring's @Transactional, we don’t manually handle begin, commit, or rollback. Spring does it for us, making transaction management clean and declarative.”

**2.What is ORM tool?**

“An ORM tool maps Java classes to database tables, allowing us to perform database operations using Java objects instead of SQL queries. Hibernate is the most widely used ORM tool in Java.”

Other ORM tools: ∙ TopLink. ∙ EclipseLink. ∙ OpenJPA. ∙ MyBatis (formally known as iBatis).

**3.What is Hibernate?**

Hibernate is a Java framework that simplifies the development of Java application to interact with the database. It is an open source, lightweight, ORM (Object Relational Mapping) tool. Hibernate implements the specifications of JPA (Java Persistence API) for data persistence.

**4.What is JPA?**

• Java Persistence API (JPA) is a Java specification that provides certain functionality and standard to ORM tools. The javax.persistence package contains the JPA classes and interfaces.

**5.EntityManagerFactory**

**6.EntityManager**

**7.Entity Transaction**

**8.** **DIFFERENCE BETWEEN PERSIST AND MERGE**

**Persist**

Persist should be called only on new entities.

• It will persist the entity object in database.

• If you pass the object with duplicate primary key it will throw exception.

**Merge**

•merge is meant to reattach detached entities.

• It will update the object in the database for duplicate key.

• If the primary key is not matched it will insert the object as new record in table.

**9.Mapping Annotations**

**\* OneToOne-Unidirectional**

🔹 Unidirectional One-to-One Mapping

* Definition: One entity maintains the relationship; the other does not reference back.
* Owning Side: The entity that contains the foreign key column.
* Annotation Usage:
  + Use @OneToOne on the owning side.
  + Use @JoinColumn(name = "foreign\_key\_column") to specify where the foreign key is stored.
* Example:

java

@Entity

public class Pan {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

private String address;

private String panNumber;

@OneToOne

@JoinColumn(name = "person\_id") // Foreign key in Pan table

private Person person;

}

@Entity

public class Person {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

private String name;

private String email;

private long phone;

}

* Database Effect:
  + The Pan table contains person\_id as a foreign key.
  + The Person table has no reference to Pan.

**🔹 Bidirectional One-to-One Mapping**

* Definition: Both entities reference each other, creating a two-way relationship.
* Owning Side: The entity without mappedBy annotation (the one storing the foreign key).
* Inverse Side: The entity with mappedBy, indicating the relationship is controlled elsewhere.
* Annotation Usage:
  + Use @OneToOne on both entities.
  + Use mappedBy = "fieldName" on the inverse side.
* Example:

java

@Entity

public class Pan {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

private String address;

private String panNumber;

@OneToOne

@JoinColumn(name = "person\_id") // Foreign key stored in Pan table

private Person person;

}

@Entity

public class Person {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

private String name;

private String email;

private long phone;

@OneToOne(mappedBy = "person") // Pan is the owning side

private Pan pan;

}

* Database Effect:
  + The Pan table contains person\_id as a foreign key.
  + The Person table does not contain pan\_id (it refers back using mappedBy).

**Key Differences**

| Mapping Type | Owning Side | Uses mappedBy? | Foreign Key Stored In |
| --- | --- | --- | --- |
| Unidirectional | The referencing entity (e.g., Pan) | ❌ No | Owning Entity's Table (Pan table) |
| Bidirectional | The entity without mappedBy (e.g., Pan) | ✅ Yes (on inverse side) | Owning Entity's Table (Pan table) |