**✅ What is ORM?**

**ORM** stands for **Object Relational Mapping**.

It is the process of **linking database tables** with **Java classes (Business Objects - BO)**.  
Each **DB table column** is mapped to a **Java class property**, and both stay **in sync** with each other.

This linking process is called **"ORM (Object Relational Mapping)"**.

**🔁 What is Synchronization in ORM?**

If you **modify the object** of a Java class:

* That change is **reflected in the database table**.

Similarly, if you **update the database**, the changes can be reflected back to the Java object.

**🛠️ Common ORM Tools**

* Hibernate
* iBatis (now MyBatis)
* EclipseLink
* ...and others

### 🌱 What is Spring ORM?

* ❌ **Not a new ORM framework**  
  Spring ORM is **not** another ORM like Hibernate or iBatis.

• It helps you use other ORM tools like Hibernate or iBatis more easily by hiding the complex setup and code, so you don’t have to write everything by yourself.

* It helps you **connect Java classes to database tables** without writing complicated code, making your work **faster and simpler**.
* 🧱 **Provides helpful template classes**  
  For example, HibernateTemplate is a helper class that **removes boilerplate code** (like opening/closing sessions and transactions).

**🧾 Plain Hibernate (without Spring ORM)**

To save or update data using Hibernate, you normally follow these steps:

1. ✅ **Create a Configuration object**  
   – This activates Hibernate with settings like DB details.
2. ✅ **Create a SessionFactory object**  
   – Used to create sessions (like a connection factory).
3. ✅ **Create a Session object**  
   – Opens a connection to the database.
4. ✅ **Begin a transaction**  
   – Starts the DB operation safely.
5. 🛠️ **Do the persistence operation**  
   – e.g., save, update, delete, etc.
6. ✅ **Commit or rollback the transaction**  
   – Saves changes or cancels if there's an error.
7. ✅ **Close session and session factory**  
   – Releases memory/resources.

**🔁 Boilerplate Code**

Steps **1, 2, 3, 4, 6, and 7** are common and repeated every time — this is called **"boilerplate code"**.

### 🔁 OR-Mapping Persistence Logic – Execution Flow (from top to bottom)

1. **OR-Mapping Persistence Logic**  
   ➤ This is your application logic where you work with Java objects and want to save or retrieve them from a database.
2. **Spring Core + Spring ORM**  
   ✅ Instead of manually writing all Hibernate code, you just **inject HibernateTemplate** using Spring.  
   ✅ Spring handles most of the boilerplate code for you.  
   ✅ You only focus on business logic.
3. **Plain ORM (Hibernate)**  
   ➤ Hibernate performs ORM — it translates your Java object changes into SQL queries.
4. **Plain JDBC**  
   ➤ Under the hood, Hibernate uses JDBC to communicate with the database.
5. **Database**  
   ➤ Final step where your data is saved or fetched.

### ✅ Advantages of Spring ORM (Made Easy)

1. **No need to write repeated code**  
   Spring ORM gives **ready-made template classes** (like HibernateTemplate) so you don't have to write the same setup code every time.
2. **Handles exceptions in a simple way**  
   You only need to handle one **common exception** (DataAccessException) instead of many different exceptions from each ORM tool.
3. **Works with any database or ORM tool**  
   Your code can easily run on **different databases** and with **different ORM tools** without much change.
4. **Supports transaction management**  
   It provides built-in support for **starting, committing, and rolling back transactions**, making your code safer and cleaner.
5. **Supports standard query operations**  
   You can use **JPA-style queries** like JPQL (Java Persistence Query Language) and **standard operations** to fetch or update single records.

Why is only DTO used from Controller to Service?

* DTO (Data Transfer Object) is like a simple box for carrying data from the user or front-end to the backend.
* The Controller gets data from the user and puts it in a DTO.
* The Service receives this DTO and uses it. This keeps the user data separate from business logic.
* Using DTOs makes sure the backend is not directly tied to how the front-end or API sends data.

Why is BO used from Service to DAO and DAO to DB?

* BO (Business Object) represents the main business data and rules of the application.
* The Service takes the DTO and turns it into a BO. Here, business logic and validation are applied.
* The DAO (Data Access Object) uses the BO to interact with the database.
* BOs make sure the database operations work with real business data, not just simple transfer data.

Why not use DTO everywhere?

* DTOs are only for moving data between the outside world (like UI or API) and the backend.
* BOs are for real business processing and database storage.
* Keeping them separate makes the application easier to manage, more secure, and less error-prone.

**✅ Why is BO (Business Object) used from DB → DAO → Service?**

* When we get data from the database, we want to bring it back into our application in a **meaningful way**.
* **BO** holds **actual business data** like salary, name, or ID — exactly how our application needs it.
* **DAO** reads the database and puts the result into a **BO** — this BO represents **real business things**.
* The **Service layer** works on this BO to apply rules, calculations, or checks as needed.
* Using BOs ensures that the data coming from the database is **clean, useful, and ready** for business processing.

**✅ Why is DTO (Data Transfer Object) used from Service → Controller?**

* After processing the business logic, we want to **send only the required output** to the controller.
* The **Service layer** takes the BO and converts it into a **DTO**.
* A **DTO** is simple — it only contains the **exact fields** that the controller or API needs to return.
* This helps to **hide internal logic** and **protect sensitive data**.
* Using DTOs makes the response **clean, safe, and easier to understand** for the UI or client.

**✅ Why is VO (View Object) used from Controller → View (UI)?**

* Once the controller gets the data, it prepares it for the user interface or webpage.
* A **VO** is used here — it is specially designed to show data on the **screen**.
* VO may include **extra formatting** (like currency symbols, date formats) that are only for display.
* It helps make the **UI look nice and user-friendly** without affecting backend code.
* VO ensures that the UI gets only what it needs — in a **display-ready format**.

Eg: SpringOrmUsingXml

**1. Load Spring container**

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

* Loads the Spring configuration from applicationContext.xml.
* All beans (dataSource, sessionFactory, transactionManager, studentDao, studentService) are created and wired by Spring.
* This initializes the entire application backend without writing new keyword manually.

**🔹 2. Get the service bean from the container**

StudentService service = (StudentService) context.getBean("studentService");

* Retrieves the studentService bean, which is of type StudentServiceImpl.
* Spring already injected the DAO bean (studentDao) into this service via setter injection:

<bean id="studentService" class="in.orcas.service.StudentServiceImpl">

<property name="dao" ref="studentDao"/>

</bean>

* Internally, this invokes: setDao(StudentDAOImpl dao).

**🔹 3. Create a StudentDTO object and populate data**

StudentDTO dto = new StudentDTO();

dto.setId(1001);

dto.setName("Ravi");

dto.setMarks(78);

* This is a simple **Data Transfer Object**.
* It carries user input data to the service layer.
* It’s not connected to the database or any persistence logic.
* This is useful for decoupling client code from persistence layer.

**🔹 4. Call the saveStudent() method**

service.saveStudent(dto);

Inside StudentServiceImpl.saveStudent():

java

CopyEdit

StudentBO bo = new StudentBO();

bo.setId(dto.getId());

bo.setName(dto.getName());

bo.setMarks(dto.getMarks());

dao.save(bo);

* Converts the DTO into a **Business Object (BO)**.
* Why? Because StudentBO is mapped to the DB table via Hibernate.
* Then it calls the DAO method dao.save(bo);.

Inside StudentDAOImpl.save():

Session session = sessionFactory.getCurrentSession();

session.save(bo);

* Gets the Hibernate session from SessionFactory (which is injected by Spring).
* session.save(bo) issues an SQL INSERT statement to Oracle DB and adds the student record in student14 table.

**🔹 5. Update student data**

dto.setName("Ravi Kumar");

dto.setMarks(85);

service.updateStudent(dto);

Inside StudentServiceImpl.updateStudent():

* Again converts DTO → BO.
* Calls dao.update(bo);.

Inside StudentDAOImpl.update():

Session session = sessionFactory.getCurrentSession();

session.update(bo);

* Issues an SQL UPDATE to change name and marks in DB.
* The update is applied to the student record with ID = 1001.

**🔹 6. Fetch all students from DB**

List<StudentDTO> list = service.getAllStudents();

Inside StudentServiceImpl.getAllStudents():

List<StudentBO> bos = dao.fetchAll();

Inside StudentDAOImpl.fetchAll():

return hibernateTemplate.loadAll(StudentBO.class);

* Uses HibernateTemplate to run SELECT \* FROM student14.
* Returns list of BOs.

Back in StudentServiceImpl:

for (StudentBO bo : bos) {

StudentDTO dto = new StudentDTO();

dto.setId(bo.getId());

dto.setName(bo.getName());

dto.setMarks(bo.getMarks());

dtos.add(dto);

}

* Converts each BO → DTO.
* Returns a clean, detached list to the UI.

**🔹 7. Print all students**

for (StudentDTO s : list) {

System.out.println(s.getId() + " " + s.getName() + " " + s.getMarks());

}

* Iterates over the list of StudentDTO and prints each record.
* Output looks like:

1001 Ravi Kumar 85

**🔹 8. Fetch student by ID**

StudentDTO student = service.getStudentById(1001);

Inside StudentServiceImpl.getStudentById():

StudentBO bo = dao.fetchById(id);

Inside StudentDAOImpl.fetchById():

return hibernateTemplate.get(StudentBO.class, id);

* Runs: SELECT \* FROM student14 WHERE id = 1001
* Returns matching BO, which is converted to DTO.

Print:

System.out.println("Fetched by ID: " + student.getId() + " " + student.getName() + " " + student.getMarks());

* Output:

Fetched by ID: 1001 Ravi Kumar 85

**🔹 9. Delete student by ID**

service.deleteStudent(1001);

Inside StudentServiceImpl.deleteStudent():

dao.delete(id);

Inside StudentDAOImpl.delete():

StudentBO bo = session.get(StudentBO.class, id);

if (bo != null) {

session.delete(bo);

}

* Hibernate issues:

DELETE FROM student14 WHERE id = 1001;

**🔹 10. Verify deletion by fetching again**

List<StudentDTO> remainingStudents = service.getAllStudents();

System.out.println("Remaining Students: ");

for (StudentDTO s : remainingStudents) {

System.out.println(s.getId() + " " + s.getName() + " " + s.getMarks());

}

* Same process as Step 6.
* This time, the list is empty or shows other students (if any).