CS-29: Advance Java Programming (J2EE)

B.C.A. (Honours) & B.C.A. (Honours with Research) — Semester 5 & 6

Saurashtra University | Effective from June 2025

Complete Classroom Notes — Theory + Diagrams + Examples + Exam Q&A

Prepared By **Alpesh Sir**
Shree Mahila College – Khamta

Table of Contents

Objectives of this course:

- Gain a deep understanding of J2EE architecture — Servlets, JSP.
- Build proficiency in Spring Framework, Hibernate, Spring Boot.
- Learn the MVC design pattern and apply it in real J2EE projects.

Prerequisite: Core Java (Classes, Objects, Exception Handling, Collections).

UNIT 1 — Introduction to J2EE and JDBC

1.1 What is J2EE?

> **Definition:** J2EE (Java 2 Enterprise Edition), now called **Java EE / Jakarta EE**, is a platform built on top of Core Java (J2SE) used to develop **large-scale, distributed, multi-tier, web-based enterprise applications**. It provides a set of specifications (Servlet, JSP, EJB, JMS, JavaMail, JSF, JNDI) and a runtime (Application Server) to run business applications.

Think of Core Java (J2SE) as the engine of a car, and J2EE as the full vehicle body built to carry passengers (business data) safely from one place to another (client to database) using standard parts (APIs).

Ordinary desktop programs written in Core Java run on a single machine and serve one user at a time. Real businesses (banks, colleges, e-commerce sites), however, need software that can be accessed by thousands of users simultaneously, over a network, with guaranteed security, transaction safety, and scalability. J2EE was created precisely to solve this problem: it standardizes how a Java application talks to a web browser, how it manages database transactions, how it authenticates users, and how it scales across multiple servers — so that every vendor's application server (Tomcat, WebLogic, JBoss, GlassFish) understands and runs the same J2EE-compliant code. Because J2EE is a *specification* (a set of rules), and not a single product, a developer who learns the J2EE APIs can move an application between different servers with minimal changes — this is called **"Write Once, Run Anywhere"** extended to the enterprise level.

J2EE applications are usually **multi-tiered** and **component-based**: the UI is built from Servlets/JSP components, the business logic from EJB/Spring components, and persistence from JDBC/Hibernate components — each piece can be developed, tested, and replaced independently, which is what makes large college-management systems or banking systems maintainable over many years.

Board Note — J2EE in one line

J2EE = Core Java + Enterprise APIs (Servlet, JSP, EJB...) + Application Server → to build Web/Enterprise Applications

1.2 Enterprise Architecture Styles

"Architecture" here means *how the different parts of an application are arranged and how they communicate with each other.* The choice of architecture decides how easy the system is to maintain, how many users it can support, and how securely data is handled. As applications grew from small single-user tools into large multi-user enterprise systems, the architecture evolved through three broad stages — Two-Tier, Three-Tier, and N-Tier — each solving the scalability and maintainability problems of the previous one.

1.2.1 Two-Tier Architecture (Client-Server)

The client directly talks to the database server. There is no separate business logic layer — logic is written either in the client or inside the database (stored procedures).

This is the simplest possible architecture and was common in early desktop database applications (e.g. a VB or Java Swing application connecting directly to MS Access/MySQL). Every client machine needs its own direct connection to the database, and any change in business rule (say, a new discount policy) means updating and redistributing the client application to every single user's machine — which does not scale well once the number of users grows into hundreds or thousands.

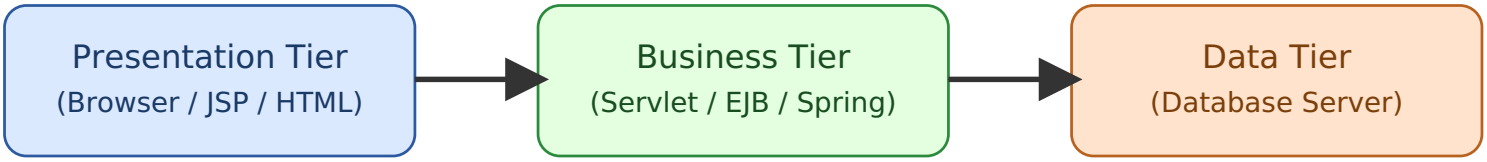
Drawback: Business logic mixed with UI → hard to maintain, no reusability, not scalable for many users.

Real classroom example: Imagine a small library-management Java Swing application installed on 20 computers in a computer lab, where each computer's program directly opens a JDBC connection to a central MySQL database and executes SQL to issue/return books. If the library later decides to add a new rule ("a student cannot issue more than 3 books at a time"), that rule must be coded and re-installed on all 20 machines individually — this is the core weakness of the two-tier model that motivated the shift to three-tier and N-tier designs, where such a rule would live in one central Business Tier used by everyone.

1.2.2 Three-Tier Architecture

Separates the application into three independent layers, each with a single responsibility:

1. **Presentation Tier** — UI layer (HTML, JSP, Servlet front-end) — interacts with the user.
2. **Application/Business Tier** — Business logic layer (Servlets, EJB, Spring services) — processes rules and data.
3. **Data Tier** — Database layer (MySQL, Oracle) — stores and retrieves data.



Advantage: Each layer can be changed/scaled independently; more secure and maintainable. This is the standard architecture used in J2EE.

Because the client never touches the database directly, the Data Tier is hidden behind the Business Tier — this greatly improves security (a hacker attacking the browser cannot see database credentials) and allows the same business logic to be reused by multiple types of clients: a web browser, a mobile app, and a desktop app can all call the same Business Tier without duplicating logic. This separation is also what makes automated testing possible — the business logic can be tested independently of both the UI and the database.

1.2.3 N-Tier Architecture

An extension of three-tier where the Business Tier itself is broken into multiple layers, e.g., Presentation → Controller → Service → DAO (Data Access Object) → Database. Modern Spring Boot applications typically follow N-Tier design.

N-Tier architecture takes the "separate responsibilities" idea of three-tier and applies it more finely inside the Business Tier itself. The Controller only handles incoming requests and routing; the Service layer holds the actual business rules and validations; the DAO/Repository layer is solely responsible for talking to the database. This extra separation means, for example, that the database access code (DAO) can be swapped from MySQL to PostgreSQL without touching a single line of business logic in the Service layer — a huge advantage in large, long-lived enterprise systems.

1.3 The J2EE Platform & Introduction to J2EE APIs

API	Full Form / Purpose
Servlet	Java class that handles HTTP requests/responses on the server side.
JSP	Java Server Pages — HTML mixed with Java code to build dynamic web pages.
EJB	Enterprise Java Beans — reusable server-side business components (transactions, security).
JMS	Java Messaging Service — for asynchronous communication between applications (queues/topics).
JavaMail	API to send/receive emails from a Java application.
JSF	Java Server Faces — component-based UI framework for building web front-ends.
JNDI	Java Naming and Directory Interface — used to look up resources (DataSource, EJB) by name.

Not every J2EE application uses all of these APIs — a small college website might only need Servlet, JSP, and JDBC, while a large banking system may additionally use EJB (for transaction-safe business components), JMS (to queue transactions between systems), and JavaMail (to send account alerts). The J2EE Platform is essentially a "collection of specifications" bundled together, and an Application Server (unlike a plain Web Server) implements the *full* set of these APIs, whereas a Web Container like Tomcat implements only the web-related ones (Servlet, JSP).

CLIENT LAYER — Browser / Mobile App / Desktop App

Web Container
Servlet · JSP · JSF

EJB Container
EJB · JMS · JavaMail

JNDI — Naming & Directory Lookup Service

DATA LAYER — Database Server (via JDBC)

Board Note: The Web Container and EJB Container together, running inside one Application Server, form the complete J2EE runtime — the Web Container faces the client, the EJB Container faces the business rules, and both ultimately talk down to the database.

1.4 Containers & Tomcat as a Web Container

> A **Container** is a runtime environment provided by an Application/Web Server that manages the life cycle, security, threading, and networking of components like Servlets and JSPs, so the developer only focuses on business logic.

Apache Tomcat is the most popular open-source **Web Container** — it implements the Servlet and JSP specifications. When a servlet class is deployed on Tomcat, Tomcat creates its object, calls its lifecycle methods, and manages multiple client requests using threads.

Without a container, a developer would have to manually write code to open a network socket, listen for HTTP requests, parse the raw request text, create a thread for every client, manage security certificates, and so on — this is exactly the kind of repetitive "plumbing" work that a container takes away. This idea is sometimes called **Inversion of Control** at the infrastructure level: instead of the programmer's code calling the server, the *server (container) calls the programmer's code* (e.g. it calls `doGet()` on the servlet) whenever a matching request arrives — hence containers are also informally called the "Hollywood Principle" in action: "*Don't call us, we'll call you.*"

1.4.1 Tomcat Directory Structure (must know for practicals)

Folder	Purpose
`bin`	Startup/shutdown scripts (`startup.bat`, `shutdown.bat`)
`conf`	Server configuration files (`server.xml`, `web.xml`)
`webapps`	Deployed web applications (each folder/WAR = one app)
`lib`	Shared JAR libraries used by all deployed apps
`logs`	Server and application log files

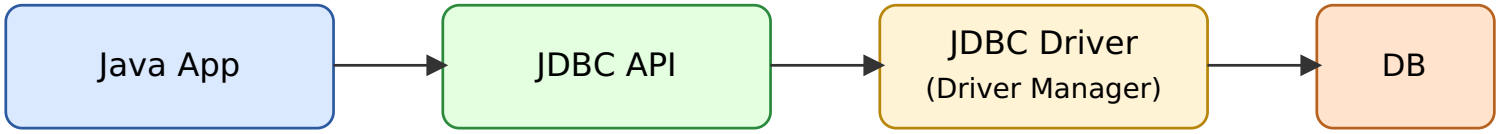
Standard Web Application Folder Structure inside `webapps/<appname>`:
`/index.jsp` (public files)
`/WEB-INF/web.xml` (deployment descriptor)
`/WEB-INF/classes/` (compiled .class files)
`/WEB-INF/lib/` (JAR dependencies)
Note: everything inside WEB-INF is protected — it cannot be accessed directly by a browser URL, only through a servlet/JSP.

1.5 JDBC (Java Database Connectivity)

> **JDBC** is a Java API that allows Java applications to connect to and execute queries on a relational database (MySQL, Oracle, etc.) in a database-independent way.

Before JDBC, every database vendor had its own proprietary API, so a program written for Oracle could not talk to MySQL without a complete rewrite. JDBC solves this by defining a standard set of Java interfaces (`Connection`, `Statement`, `ResultSet`, etc.) that every database vendor implements through its own *driver*. The application code is written once against these standard interfaces; only the driver JAR file and the connection URL change when switching databases — this is the same "program to an interface, not an implementation" principle used throughout J2EE.

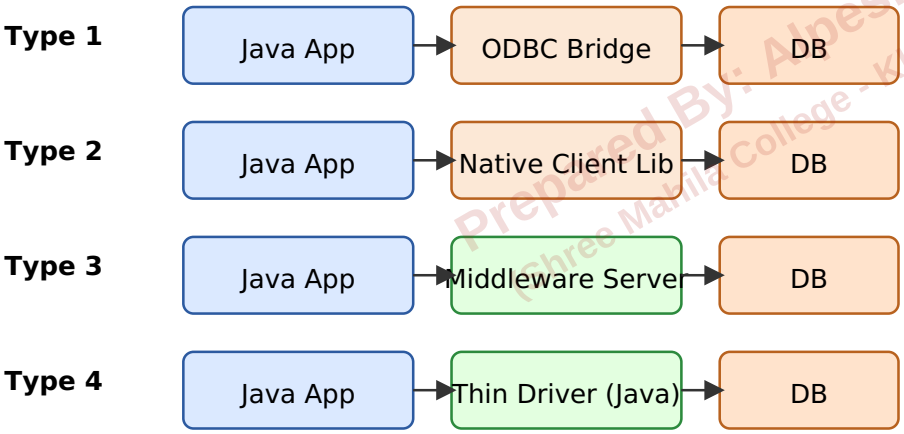
1.5.1 JDBC Architecture



Flow: Java Application → JDBC API (java.sql.*) → JDBC Driver (converts JDBC calls into database-specific calls) → Database.

1.5.2 Types of JDBC Drivers

Type	Name	Description
Type 1	JDBC-ODBC Bridge Driver	Converts JDBC calls to ODBC calls. Obsolete, platform-dependent.
Type 2	Native-API Driver	Converts JDBC calls into native database client API calls (partly Java).
Type 3	Network Protocol Driver	Sends requests to a middleware server which then talks to the DB (fully Java).
Type 4	Thin Driver	Directly converts JDBC calls into the vendor-specific DB protocol. Fully Java, fastest, most used today (e.g. MySQL Connector/J).



Type 4 (Thin Driver) is 100% Java, needs no extra native software, and is the standard choice in almo

1.5.3 Major JDBC Classes and Interfaces

Interface/Class	Purpose
`DriverManager`	Manages the list of database drivers; creates the Connection.
`Connection`	Represents a session/connection with a specific database.
`Statement`	Used to execute simple static SQL queries.
`PreparedStatement`	Precompiled SQL statement — faster, prevents SQL injection.
`CallableStatement`	Used to call stored procedures.
`ResultSet`	Holds the table-like data returned by a SELECT query.

1.5.4 Steps to Create a Simple JDBC Application

7 Steps of JDBC (must remember for exam):

1. Import JDBC packages (`java.sql.*`)

2. Load and Register the Driver
3. Create Connection using `DriverManager.getConnection()`
4. Create Statement
5. Execute the Query
6. Process the ResultSet
7. Close the Connection

1.5.5 Example — Simple JDBC SELECT Application

```
import java.sql.*;

public class JdbcSelectExample {
    public static void main(String[] args) {
        try {
            // Step 1 & 2: Load driver (auto-loaded in modern JDBC 4.0+)
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Step 3: Create Connection
            Connection con = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/college", "root", "password");

            // Step 4: Create Statement
            Statement st = con.createStatement();

            // Step 5: Execute Query
            ResultSet rs = st.executeQuery("SELECT * FROM student");

            // Step 6: Process ResultSet
            while (rs.next()) {
                System.out.println(rs.getInt("id") + " " + rs.getString("name"));
            }

            // Step 7: Close Connection
            con.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```


1.5.6 Types of Statement Interfaces — Comparison

Feature	Statement	PreparedStatement	CallableStatement
Use	Static SQL query	Parameterized (dynamic) SQL query	Calling stored procedures
Compilation	Compiled every time	Precompiled — faster on repeat execution	Precompiled procedure call
SQL Injection safe?	No	Yes	Yes
Syntax	`st.executeQuery("SELECT...")`	`con.prepareStatement("...WHERE id=?")`	`con.prepareCall("{call proc(?)}")`

PreparedStatement Example (Insert with parameters):

```
String sql = "INSERT INTO student(id,name) VALUES(?,?)";
PreparedStatement ps = con.prepareStatement(sql);
ps.setInt(1, 101);
ps.setString(2, "Rahul");
ps.executeUpdate();
```

1.5.7 CRUD Application with JDBC

CRUD = Create, Read, Update, Delete — the four basic database operations.

Operation	SQL	JDBC Method
Create	INSERT INTO ...	`executeUpdate()`

Read	SELECT * FROM ...	`executeQuery()`
Update	UPDATE ... SET ...	`executeUpdate()`
Delete	DELETE FROM ...	`executeUpdate()`

```
// UPDATE Example
PreparedStatement ps = con.prepareStatement(
    "UPDATE student SET name=? WHERE id=?");
ps.setString(1, "Amit");
ps.setInt(2, 101);
int rows = ps.executeUpdate();

// DELETE Example
PreparedStatement ps2 = con.prepareStatement("DELETE FROM student WHERE id=?");
ps2.setInt(1, 101);
ps2.executeUpdate();
```

1.5.8 Transaction Management in JDBC

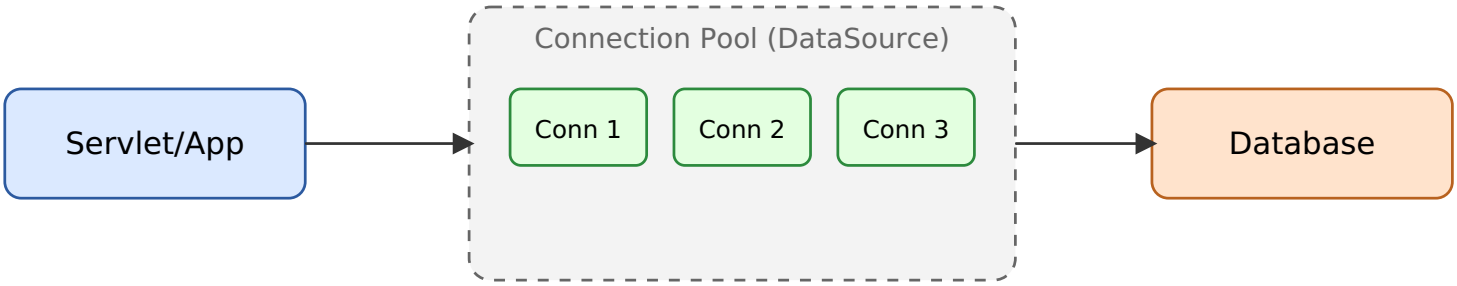
A **Transaction** is a group of one or more SQL statements that must execute together as a single unit — either *all* of them succeed (COMMIT) or *none* of them take effect (ROLLBACK). This guarantees data consistency, especially in operations like bank transfers.

By default, JDBC runs in **auto-commit mode** — every single SQL statement is committed to the database immediately after it executes. For multi-step operations (e.g. debit one account, credit another), auto-commit is dangerous because if the second statement fails, the first one has already been permanently saved, leaving the database in an inconsistent state. Turning off auto-commit lets the developer group several statements into one logical transaction.


```
con.setAutoCommit(false);   // start manual transaction
try {
    PreparedStatement ps1 = con.prepareStatement(
        "UPDATE account SET balance = balance - ? WHERE id = ?");
    ps1.setDouble(1, 5000); ps1.setInt(2, 101);
    ps1.executeUpdate();

    PreparedStatement ps2 = con.prepareStatement(
        "UPDATE account SET balance = balance + ? WHERE id = ?");
    ps2.setDouble(1, 5000); ps2.setInt(2, 102);
    ps2.executeUpdate();

    con.commit();          // both succeeded -> save permanently
} catch (Exception e) {
    con.rollback();        // any failure -> undo everything
}
```

1.5.9 Connection Pooling & DataSource

Opening a fresh database connection for every single request is expensive (it involves a network handshake, authentication, and resource allocation on the DB server) — under heavy traffic this can slow an application drastically or even exhaust the database server's connection limit. A **Connection Pool** solves this by creating a fixed number of connections in advance and reusing them: when a servlet needs a connection, it "borrows" one from the pool; when done, it "returns" it instead of closing it permanently. This is managed through a `javax.sql.DataSource` object, usually configured once in the server (or Spring Boot's `application.properties`), rather than through `DriverManager` directly in application code.



1.5.10 Complete CRUD Console Program (Exam-Important Full Program)

This single program demonstrates all four CRUD operations together using a menu-driven console application — a very common practical/exam question.

```
import java.sql.*;
import java.util.Scanner;

public class StudentCRUD {
    static Connection con;

    public static void main(String[] args) throws Exception {
        Class.forName("com.mysql.cj.jdbc.Driver");
        con = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/college", "root", "password");

        Scanner sc = new Scanner(System.in);
        int choice;
        do {
            System.out.println("\n1.Insert 2.View 3.Update 4.Delete 5.Exit");
            choice = sc.nextInt();
            switch (choice) {
                case 1: insert(sc); break;
                case 2: view(); break;
                case 3: update(sc); break;
                case 4: delete(sc); break;
            }
        } while (choice != 5);
        con.close();
    }

    static void insert(Scanner sc) throws Exception {
        System.out.print("Enter ID and Name: ");
        int id = sc.nextInt(); String name = sc.next();
        PreparedStatement ps = con.prepareStatement(
            "INSERT INTO student VALUES(?,?)");
        ps.setInt(1, id); ps.setString(2, name);
        ps.executeUpdate();
        System.out.println("Record Inserted.");
    }

    static void view() throws Exception {
        Statement st = con.createStatement();
        ResultSet rs = st.executeQuery("SELECT * FROM student");
        while (rs.next())
            System.out.println(rs.getInt(1) + " - " + rs.getString(2));
    }

    static void update(Scanner sc) throws Exception {
        System.out.print("Enter ID to update, new Name: ");
        int id = sc.nextInt(); String name = sc.next();
        PreparedStatement ps = con.prepareStatement(
            "UPDATE student SET name=? WHERE id=?");
        ps.setString(1, name); ps.setInt(2, id);
        System.out.println(ps.executeUpdate() + " row(s) updated.");
    }

    static void delete(Scanner sc) throws Exception {
        System.out.print("Enter ID to delete: ");
```

```
        int id = sc.nextInt();
        PreparedStatement ps = con.prepareStatement(
            "DELETE FROM student WHERE id=?");
        ps.setInt(1, id);
        System.out.println(ps.executeUpdate() + " row(s) deleted.");
    }
}
```

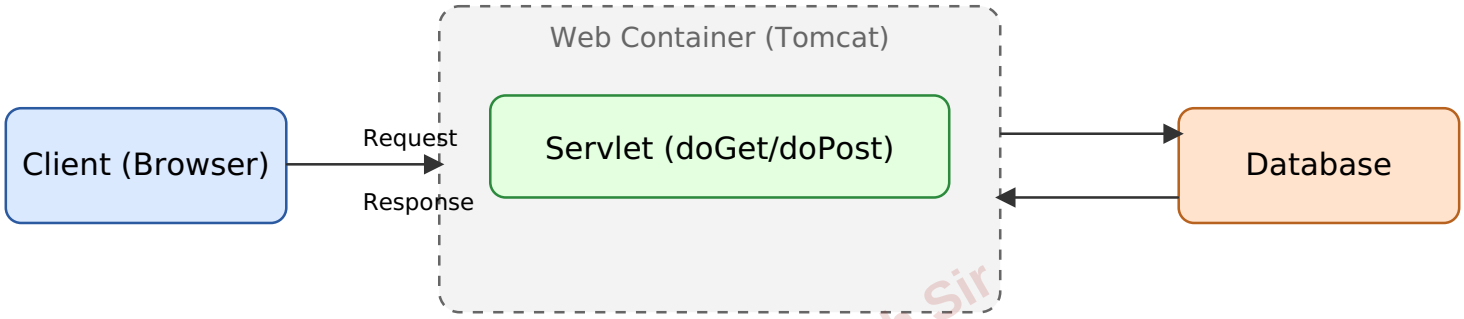
UNIT 2 — Servlet

2.1 Servlet Introduction

> A **Servlet** is a Java class that runs on a web/application server and extends the server's capability to handle client requests (usually HTTP) and generate dynamic responses. It is the server-side equivalent of an "applet."

Before servlets, dynamic web content on the server side was generated using CGI (Common Gateway Interface) scripts, where the web server would start a brand-new operating-system process for every single incoming request — this was extremely slow and did not scale. Servlets solve this problem by running as lightweight Java *threads* inside a single long-running container process: the container creates the servlet object only once and then reuses it for every subsequent request, spawning a new thread (not a new process) each time. This makes servlets far faster and more memory-efficient than CGI, and it is the foundational technology on which JSP, Spring MVC, and virtually all Java web frameworks are built.

2.2 Architecture of a Servlet

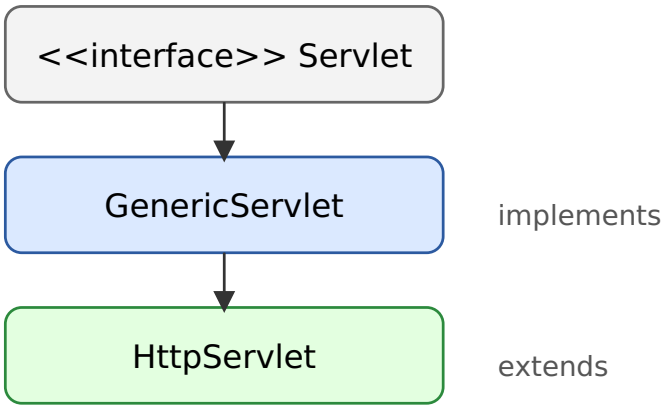


The client sends an HTTP request → Web Container receives it → creates/uses a Servlet object → Servlet processes the request (possibly hits the database) → generates a dynamic response → sent back to the client.

2.3 Servlet API — javax.servlet and javax.servlet.http

Package	Contains
`javax.servlet`	Protocol-independent interfaces: `Servlet`, `ServletConfig`, `ServletContext`, `GenericServlet`, `RequestDispatcher`.
`javax.servlet.http`	HTTP-specific classes: `HttpServlet`, `HttpServletRequest`, `HttpServletResponse`, `HttpSession`, `Cookie`.

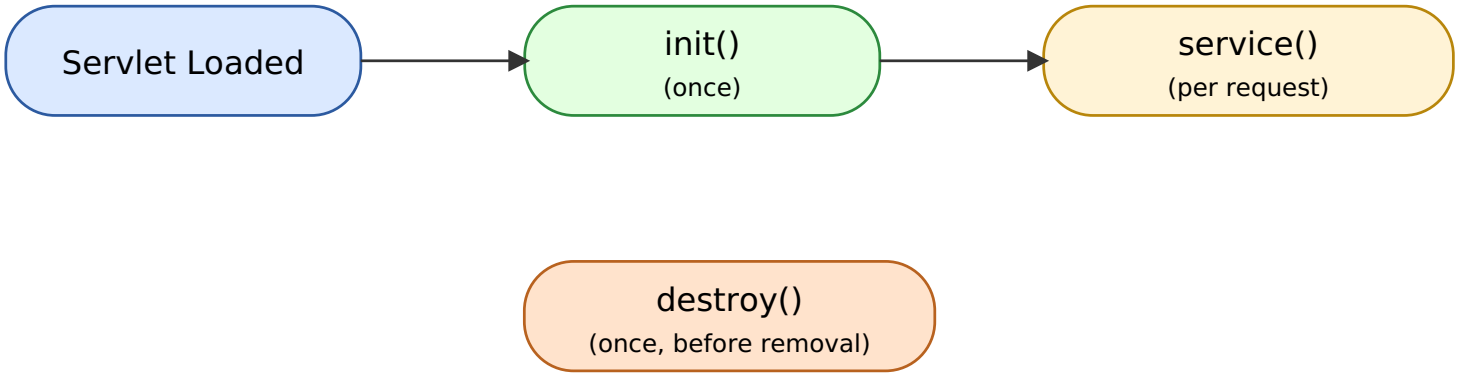
Every servlet class, directly or indirectly, implements the `Servlet` interface — but writing a class that implements `Servlet` directly means manually overriding all five of its methods (`init`, `service`, `getServletConfig`, `getServletInfo`, `destroy`). To avoid this, Java provides two convenience classes: `GenericServlet` (protocol-independent, implements most of the boilerplate) and `HttpServlet` (HTTP-specific, extends GenericServlet and adds `doGet()`, `doPost()`, `doPut()`, `doDelete()`). In real projects, developers almost always extend `HttpServlet` since nearly all servlets deal with HTTP.



2.4 Servlet Life Cycle

3 main phases (very important for exam):

init() → service() → destroy()



1. **Loading & Instantiation:** Container loads the servlet class and creates one object.
2. **init():** Called only **once** when the servlet is loaded — used for one-time setup (e.g. DB connection pool).
3. **service():** Called for **every** client request; internally dispatches to `doGet()`/`doPost()` based on HTTP method.
4. **destroy():** Called only **once** when the container shuts down or unloads the servlet — used for cleanup.

A very common exam confusion is between the constructor, `init()`, and `service()`. Remember: the container creates *only one object* of a servlet class (by default) no matter how many users hit it — that single object's `init()` runs exactly once to prepare shared resources, and then `service()` runs again and again, once per incoming request, possibly on different threads simultaneously for different users. This is why servlet code must be written carefully to avoid storing per-user data in instance variables (which would be shared/overwritten across users) — per-user data should instead be stored in local variables inside `service()`/`doGet()`, or in the HttpSession.

2.5 Servlet Configuration with Deployment Descriptor (web.xml)

```
<web-app>
  <servlet>
    <servlet-name>HelloServlet</servlet-name>
    <servlet-class>com.example.HelloServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>HelloServlet</servlet-name>
    <url-pattern>/hello</url-pattern>
  </servlet-mapping>
</web-app>
```

Modern servlets can also use the `@WebServlet("/hello")` annotation instead of web.xml.

2.6 Developing and Deploying a Servlet — Full Example

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloServlet extends HttpServlet {

    public void doGet(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<h1>Hello, " + req.getParameter("name") + "</h1>");
    }
}
```

Deployment steps: Write servlet class → Configure in web.xml or annotate with @WebServlet → Compile → Place .class in WEB-INF/classes → Deploy WAR on Tomcat → Access via URL pattern.

2.6.1 doGet() vs doPost() — Full Registration Form Example

HTML forms can submit data using either the GET method (data appended to the URL, visible, limited length, used for search/filter) or the POST method (data sent in the request body, hidden, no length limit, used for sensitive/large data like passwords or file uploads). A servlet must override the matching method to handle each.

```
<!-- register.html -->
<form action="register" method="post">
  Name: <input type="text" name="uname"><br>
  Email: <input type="text" name="email"><br>
  <input type="submit" value="Register">
</form>
```

```
@WebServlet("/register")
public class RegisterServlet extends HttpServlet {

    protected void doPost(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {
        String name  = req.getParameter("uname");
        String email = req.getParameter("email");

        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<h2>Registration Successful</h2>");
        out.println("<p>Name: " + name + "</p>");
        out.println("<p>Email: " + email + "</p>");
    }
}
```

2.7 Handling Servlet Requests and Responses

Object	Purpose	Common Methods
`HttpServletRequest`	Carries data sent by the client to the server	`getParameter()`, `getHeader()`, `getSession()`
`HttpServletResponse`	Used by the servlet to send data back to the client	`setContentType()`, `getWriter()`, `sendRedirect()`

2.8 Reading Initialization Parameters

```
<servlet>
  <servlet-name>HelloServlet</servlet-name>
  <servlet-class>com.example.HelloServlet</servlet-class>
  <init-param>
    <param-name>company</param-name>
    <param-value>Codizious</param-value>
  </init-param>
</servlet>
```

```
// Reading it inside init()
String company = getServletConfig().getInitParameter("company");
```

2.9 Session Tracking Approaches

> HTTP is a **stateless** protocol — the server does not remember previous requests from the same client. Session tracking techniques are used to maintain a "conversational state" across multiple requests.

"Stateless" means every HTTP request is treated by the server as if it is coming from a brand-new, unknown client — the server, by default, has no memory of a user having logged in on the previous request. This is a problem for real applications: once you log in to an online shopping site, you expect the server to remember you as you move from page to page (browsing products, adding to cart, checking out) without asking you to log in again on every click. Session tracking techniques were created to simulate this "memory" on top of the stateless HTTP protocol.

Technique	How it Works	Limitation
URL Rewriting	Session ID appended to every URL, e.g. `page.jsp;jsessionid=123`	URL becomes long/ugly; must be applied to every link
Hidden Form Fields	Session data stored in a hidden `<input type="hidden">` and resubmitted with every form	Works only with form-to-form navigation
Cookies	Small key-value data stored on the client's browser and sent automatically with each request	User may disable cookies; limited storage size
Session API (HttpSession)	Server creates a unique session object per user (`req.getSession()`), identified via a cookie (JSESSIONID) internally	Consumes server memory for each active user

```
// Using Session API
HttpSession session = req.getSession();
session.setAttribute("username", "Rahul");
String user = (String) session.getAttribute("username");
```

2.9.1 Complete Login Example Using HttpSession

```
@WebServlet("/login")
public class LoginServlet extends HttpServlet {
    protected void doPost(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {
        String uname = req.getParameter("uname");
        String pass  = req.getParameter("pass");

        if (uname.equals("admin") && pass.equals("1234")) {
            HttpSession session = req.getSession();
            session.setAttribute("user", uname);
            res.sendRedirect("welcome.jsp");
        } else {
            res.sendRedirect("login.html?error=1");
        }
    }
}

// welcome.jsp checks the session before showing content
<%
  if (session.getAttribute("user") == null) {
      response.sendRedirect("login.html");
  }
%>
Welcome, <%= session.getAttribute("user") %>!
```


2.10 RequestDispatcher — forward() and include()

`RequestDispatcher` is an object used inside a servlet to pass control to another resource (servlet, JSP, or HTML) **on the server side**, without the browser knowing (the URL in the browser's address bar does not change).

This is different from `response.sendRedirect()`, which sends an instruction back to the *browser* telling it to make a brand-new request to a different URL (the address bar changes, and the original request/response objects are lost). `RequestDispatcher.forward()`, in contrast, happens entirely inside the server in the same request — the original `request` object (with all its attributes) is preserved and passed along, which is exactly why MVC controllers use `forward()` to hand off to a JSP view.


```
RequestDispatcher rd = req.getRequestDispatcher("welcome.jsp");
req.setAttribute("msg", "Login Successful");
rd.forward(req, res);     // hands off request+response to welcome.jsp

// rd.include(req, res);  // alternative: includes the target's output, then returns control
```

2.11 Servlet Filters

A **Filter** is a component that intercepts a request **before** it reaches a servlet (or a response before it reaches the client) to perform pre-processing/post-processing tasks like authentication checks, logging, compression, or input validation — without changing the target servlet's code.

Filters implement the same "cross-cutting concern" idea we will see again with Spring AOP in Unit 5: instead of writing authentication-checking code inside every single servlet, a single Filter can be configured (via a URL pattern) to run automatically in front of many servlets, keeping that logic in one reusable place.


```
@WebFilter("/secure/*")
public class AuthFilter implements Filter {
    public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
            throws IOException, ServletException {
        HttpServletRequest hreq = (HttpServletRequest) req;
        if (hreq.getSession().getAttribute("user") == null) {
            ((HttpServletResponse) res).sendRedirect("login.html");
        } else {
            chain.doFilter(req, res);   // allow request to proceed
        }
    }
}
```

UNIT 3 — JSP (Java Server Pages)

3.1 Introduction to JSP

> **JSP (Java Server Page)** is a technology that allows HTML to be mixed with Java code to create dynamic web content. It is internally converted (by the container) into a Servlet before execution.

JSP was introduced by Sun Microsystems to solve a practical problem with pure servlets: writing an entire HTML page's worth of markup using repeated `out.println("...")` statements inside Java code is tedious, error-prone, and impossible for a web designer (who knows HTML/CSS but not Java) to work with. JSP flips this around — a page is written mainly in familiar HTML, with small islands of Java code inserted only where dynamic content is genuinely needed. This division of labor allows a Java developer to focus on logic while a designer focuses on look-and-feel, and it is why JSP became the standard "View" technology in J2EE MVC applications.

3.2 JSP vs Servlet

Servlet	JSP
Java code with embedded HTML (via println)	HTML code with embedded Java
Good for business logic	Good for presentation/UI
Must be compiled manually before deployment	Auto-compiled into a servlet by the container on first request
Harder to design UI	Easy to design UI — designer-friendly

3.3 JSP Architecture & Life Cycle

> **JSP Life Cycle methods (parallel to Servlet):**
> jspInit() → _jspService() (auto-generated, per request) → jspDestroy()

This translation happens automatically and only *once* — the first time a JSP page is requested, the container converts it into a `.java` servlet source file, compiles it into a `.class` file, and loads it, which is why the very first hit on a JSP page is noticeably slower than every subsequent hit (which simply reuses the already-compiled servlet class). If the developer edits the `.jsp` file again, the container detects the change and automatically re-translates and re-compiles it on the next request — this is what makes JSP development fast: no manual compile step is needed during development.

3.4 JSP Elements

Every JSP page is built from three broad categories of special tags, each serving a different purpose: **Directives** configure the page as a whole (before any code runs), **Scripting Elements** embed actual Java logic directly into the generated servlet, and **Action Elements** perform a predefined task (like including another page or working with a bean) using XML-style tags instead of raw Java code. Understanding which category a tag belongs to makes it much easier to remember its syntax and purpose.

3.4.1 Directive Elements — give instructions to the container

Directive	Syntax	Purpose
page	`<%@ page import="java.util.*" %>`	Page-level settings: imports, error page, content type
include	`<%@ include file="header.jsp" %>`	Static include at translation time (merges source code)
taglib	`<%@ taglib uri="..." prefix="c" %>`	Declares a custom/standard tag library (e.g. JSTL)

3.4.2 Scripting Elements — embed Java logic

Type	Syntax	Purpose
Declaration	`<%! int count = 0; %>`	Declares variables/methods (becomes class member)

Scriptlet	`<% int x = 10; %>`	Java code block (goes inside _jspService)
Expression	`<%= x+5 %>`	Outputs a value directly to the page (no semicolon)

3.4.3 Action Elements — perform an action at request time

Action	Purpose
`<jsp:param>`	Passes an extra parameter to an included/forwarded page
`<jsp:include>`	Dynamically includes another page at request time
`<jsp:forward>`	Forwards the request to another resource (server-side)
`<jsp:plugin>`	Embeds an applet/bean using browser plugin (legacy)
`<jsp:useBean>`	Instantiates or looks up a JavaBean
`<jsp:setProperty>`	Sets a property value on a bean
`<jsp:getProperty>`	Reads/prints a property value from a bean

```
<jsp:useBean id="student" class="com.example.Student" />
<jsp:setProperty name="student" property="name" value="Rahul" />
Name: <jsp:getProperty name="student" property="name" />
```

3.5 JSP Implicit Objects

Objects automatically available inside every JSP page — no need to declare them.

These objects exist because, remember, a JSP page is secretly a servlet — and a servlet's `service()` method always receives a request and response object, has access to the session and application context, and so on. Rather than forcing the developer to manually declare and cast these objects on every page, the JSP container automatically generates the necessary local variables in the translated servlet code, ready to use straight away. This is one of the biggest productivity benefits of JSP over writing raw servlets.

Object	Type	Use
request	HttpServletRequest	Read client request data
response	HttpServletResponse	Send response back to client
out	JspWriter	Write output to the page
session	HttpSession	Store per-user data
application	ServletContext	Store app-wide (shared) data
pageContext	PageContext	Access all scopes from one object

3.6 JSP Scope

Scope	Lifetime
page	Only within the current page/request cycle
request	Until the request finishes (survives forward)
session	Until the user's session ends
application	Until the application/server shuts down (shared by all users)

3.7 Including & Forwarding from JSP Pages

	include Action	forward Action
Control returns?	Yes, back to original page	No, control passes permanently
Response buffer	Appended	Cleared and replaced
Syntax	`<jsp:include page="footer.jsp"/>`	`<jsp:forward page="result.jsp"/>`

3.8 Working with Session & Cookie in JSP

```
<%
  Cookie c = new Cookie("username", "Rahul");
  response.addCookie(c);
  session.setAttribute("loginTime", new java.util.Date());
%>
```

3.9 Error Handling and Exception Handling with JSP

```
<%@ page isErrorPage="true" %>
Error occurred: <%= exception.getMessage() %>
```

```
<%@ page errorPage="error.jsp" %>  <%-- in the source page --%>
```

3.10 JSP EL (Expression Language) & JSTL

EL simplifies data access without Java scriptlets: `${student.name}`.
JSTL (JSP Standard Tag Library) provides ready-made tags for loops, conditions, etc., removing the need for scriptlets entirely.

Heavy use of scriptlets (raw `<% ... %>` Java code) inside a JSP page came to be seen as bad practice, because it mixes business logic with presentation markup again — exactly the problem JSP was meant to avoid. EL and JSTL were introduced together to let a JSP page stay "pure HTML-like" while still displaying dynamic data and even performing loops/conditions, entirely through tags and simple expressions instead of embedded Java. Modern J2EE coding standards strongly recommend EL + JSTL over scriptlets for all real-world JSP pages.


```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<c:forEach var="s" items="${studentList}">
   ${s.name}<br/>
</c:forEach>

<c:if test="${marks >= 40}">
   Pass
</c:if>
```

3.10.1 Full Working Example — Student Result Page (Servlet + JSP + EL/JSTL)

This exam-favourite example shows the complete MVC-style flow: a Servlet (Controller) fetches data and forwards it, and a JSP (View) displays it purely using EL/JSTL — no scriptlets at all.

```
// ResultServlet.java (Controller)
@WebServlet("/result")
public class ResultServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {
        List<Student> list = new ArrayList<>();
        list.add(new Student("Rahul", 78));
        list.add(new Student("Amit", 35));

        req.setAttribute("studentList", list);
        req.getRequestDispatcher("result.jsp").forward(req, res);
    }
}
```

```
<%-- result.jsp (View) --%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<table border="1">
<tr><th>Name</th><th>Marks</th><th>Status</th></tr>
```

```
<c:forEach var="s" items="${studentList}">
  <tr>
    <td>${s.name}</td>
    <td>${s.marks}</td>
    <td>
      <c:choose>
        <c:when test="${s.marks >= 40}">Pass</c:when>
        <c:otherwise>Fail</c:otherwise>
      </c:choose>
    </td>
  </tr>
</c:forEach>
</table>
```

Notice that `result.jsp` contains **zero** Java scriptlet code — this is the modern, recommended style of writing JSP pages, keeping the View strictly about presentation while the Servlet Controller does all the data preparation.

UNIT 4 — EJB, MVC Architecture, Hibernate

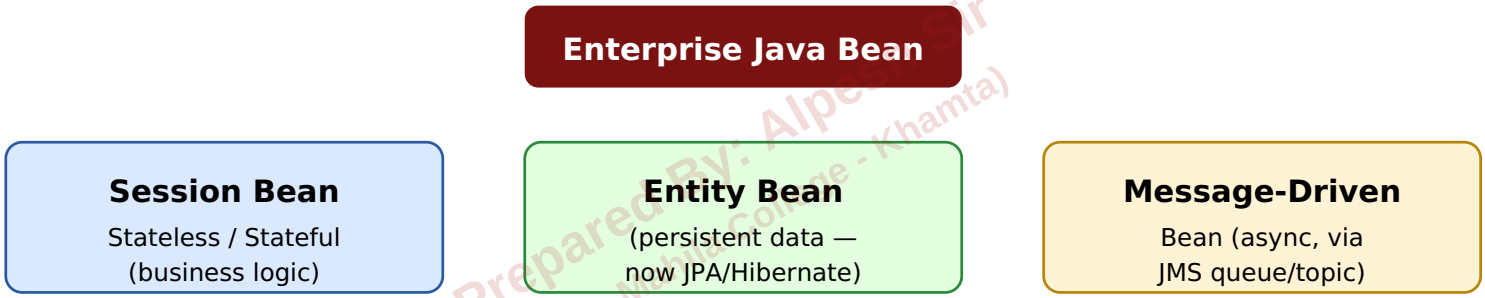
4.1 Introduction to EJB (Enterprise Java Beans)

EJB is a server-side managed component that encapsulates business logic of an enterprise application. The EJB container automatically handles transactions, security, and concurrency, so developers focus only on business rules.

Consider a bank transfer operation: money must be deducted from one account and added to another as a single, indivisible unit of work — if the deduction succeeds but the addition fails, the transaction must roll back completely to avoid losing money. Writing this transaction-management, concurrency-control, and security-checking code by hand for every business method is repetitive and error-prone. EJB was designed so that the developer only writes the actual business method (e.g. `transferFunds()`), and simply annotates or configures it — the EJB container then automatically wraps it with the transaction, security, and thread-safety code behind the scenes. This is the same "container does the plumbing" idea seen earlier with Servlets, just applied to business logic instead of web requests.

4.1.1 Types of EJB

Type	Description
Session Bean	Performs business logic; two kinds: • *Stateless* — does not retain client data between calls (fast, scalable) • *Stateful* — retains conversational state across method calls for one client
Entity Bean	Represents persistent data (a row in a database) — largely replaced by JPA/Hibernate entities today.
Message-Driven Bean (MDB)	Responds asynchronously to messages received via JMS.



4.1.2 Stateless Session Bean — Example

```
@Stateless
public class CalculatorBean {
    public int add(int a, int b) {
        return a + b;
    }
}

// Client code (via dependency injection)
@EJB
CalculatorBean calc;
int sum = calc.add(5, 10);
```

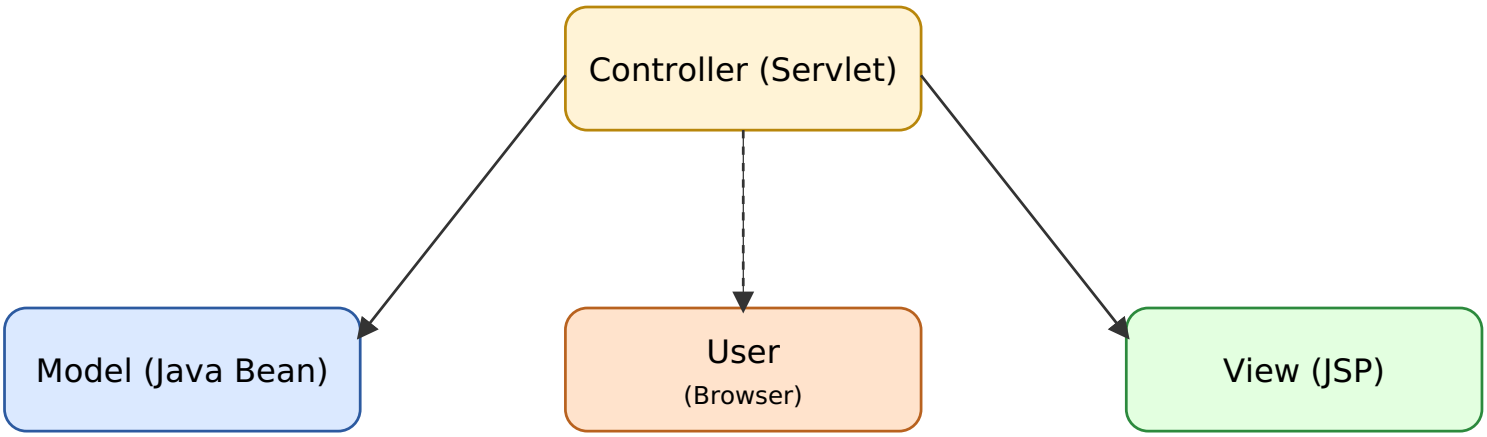
Because a Stateless bean does not remember anything about the client between method calls, the container can freely reuse the same bean instance for many different clients (pooling them for efficiency), which is why Stateless Session Beans are the fastest and most scalable EJB type — ideal for simple, self-contained operations like a calculator, a price lookup, or a stateless validation check.

4.2 Introduction to MVC Architecture

MVC (Model-View-Controller) is a design pattern that separates an application into three interconnected parts so each can be developed and tested independently.

Before MVC became standard practice, it was common to see JSP pages that contained large blocks of Java scriptlet code doing database access, business calculations, and HTML rendering all mixed together — commonly nicknamed "Model 1" architecture. This quickly became

unmanageable as applications grew: a small change in business logic could accidentally break the page layout, and vice versa. MVC (sometimes called "Model 2" architecture in the J2EE world) fixes this by strictly assigning one job to each component, and by making the Controller the single entry point that decides which Model logic to run and which View to display — this makes large applications far easier to debug, extend, and test.



Component	Responsibility	J2EE Technology used
Model	Represents data + business logic	JavaBeans, EJB, Hibernate Entities
View	Displays data to the user (UI)	JSP, HTML
Controller	Receives requests, invokes Model, chooses View	Servlet

Flow: User sends request → Controller (Servlet) receives it → Controller talks to Model (business logic/DB) → Controller forwards data to View (JSP) → View renders the response to the User.

4.3 Introduction to Hibernate

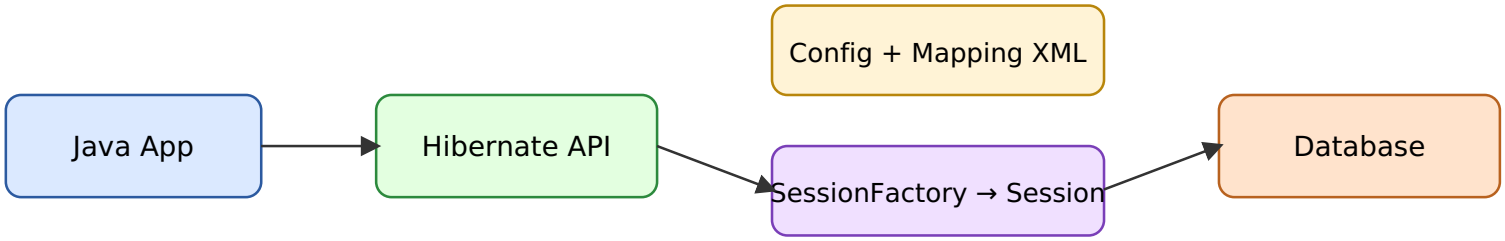
> **Hibernate** is an open-source **ORM (Object Relational Mapping)** framework for Java that maps Java classes to database tables automatically, eliminating most manual JDBC and SQL code.

There is a well-known mismatch between how Java programs think about data (as objects with fields and relationships — a `Student` object having a `List<Course>`) and how relational databases store data (as flat rows and columns spread across multiple linked tables) — this is called the **object-relational impedance mismatch**. With plain JDBC, the developer must manually write SQL to convert between these two worlds for every single table, which is repetitive and hard to maintain as the schema grows. Hibernate solves this by letting the developer work purely with Java objects — Hibernate itself generates the correct SQL behind the scenes, tracks changes to objects, and synchronizes them with the database automatically.

4.3.1 Features of Hibernate

- Open source and lightweight
- Database independent (works with MySQL, Oracle, PostgreSQL... by just changing a dialect)
- Automatic table creation from Java classes
- Caching support for better performance
- HQL — object-oriented query language

4.3.2 Hibernate Architecture



4.3.3 Hibernate Configuration File (hibernate.cfg.xml)

```
<hibernate-configuration>
 <session-factory>
  <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
  <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/college</property>
  <property name="hibernate.connection.username">root</property>
  <property name="hibernate.connection.password">password</property>
  <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
  <mapping resource="Student.hbm.xml"/>
 </session-factory>
</hibernate-configuration>
```

4.3.4 Hibernate Mapping File (Student.hbm.xml)

```
<hibernate-mapping>
 <class name="com.example.Student" table="student">
   <id name="id" column="id"/>
   <property name="name" column="name"/>
 </class>
</hibernate-mapping>
```

4.3.5 Hibernate Annotation (modern alternative to XML mapping)

```
@Entity
@Table(name="student")
public class Student {
    @Id
    @GeneratedValue
    private int id;

    @Column(name="name")
    private String name;
    // getters and setters
}
```


4.3.6 HQL (Hibernate Query Language)

HQL is object-oriented — it queries **Java class names and properties**, not table/column names.

```
Query q = session.createQuery("FROM Student WHERE name='Rahul'");
List<Student> list = q.list();
```

4.3.7 Hibernate Sessions

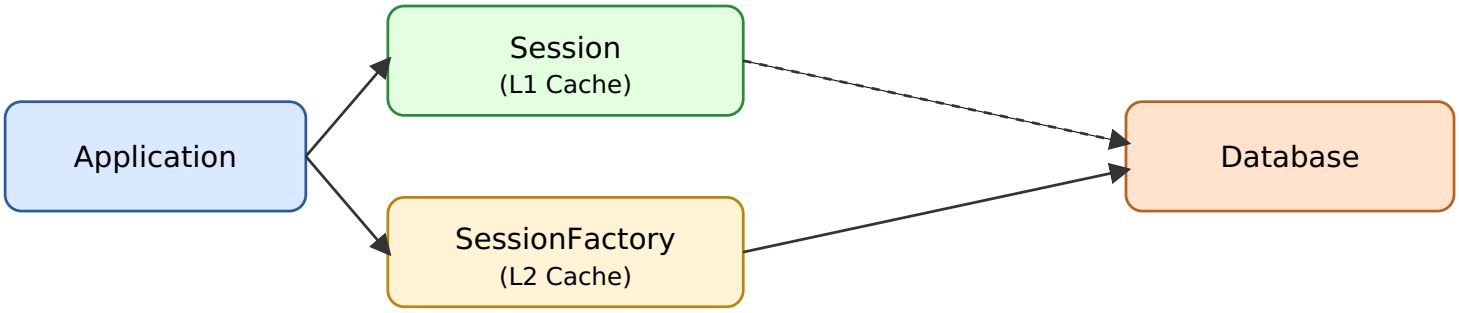
```
SessionFactory factory = new Configuration().configure().buildSessionFactory();
Session session = factory.openSession();
Transaction tx = session.beginTransaction();

Student s = new Student();
s.setName("Rahul");
session.save(s);

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

Session is a lightweight, single-threaded object representing a conversation between the application and the database — used for CRUD operations (save, get, update, delete).

4.3.8 Hibernate Caching (L1 & L2 Cache)

Every time Hibernate has to go to the database to fetch an object, it costs time (network round-trip + SQL execution). Caching stores previously-fetched objects in memory so repeated requests for the same data can be served instantly without hitting the database again.

Cache Level	Scope	Enabled By Default?
L1 (First-Level) Cache	Tied to a single `Session` — cleared when the session closes	Yes, always on, cannot be disabled
L2 (Second-Level) Cache	Tied to the `SessionFactory` — shared across all sessions/users	No, must be explicitly configured (e.g. with EhCache)



4.3.9 Complete Hibernate CRUD DAO Example

A very common exam/practical requirement is a Hibernate DAO (Data Access Object) class performing all CRUD operations for one entity, shown together below.


```
public class StudentDAO {
    SessionFactory factory = new Configuration().configure().buildSessionFactory();

    public void save(Student s) {
        Session session = factory.openSession();
        Transaction tx = session.beginTransaction();
        session.save(s);
        tx.commit();
        session.close();
    }

    public Student get(int id) {
        Session session = factory.openSession();
        Student s = session.get(Student.class, id);
        session.close();
        return s;
    }

    public void update(Student s) {
        Session session = factory.openSession();
        Transaction tx = session.beginTransaction();
        session.update(s);
        tx.commit();
        session.close();
    }

    public void delete(int id) {
        Session session = factory.openSession();
        Transaction tx = session.beginTransaction();
        Student s = session.get(Student.class, id);
        session.delete(s);
        tx.commit();
        session.close();
    }

    public List<Student> getAll() {
        Session session = factory.openSession();
        List<Student> list = session.createQuery("FROM Student", Student.class).list();
        session.close();
        return list;
```

```
    }
}
```

UNIT 5 — Spring Framework & Spring Boot

5.1 What is Spring Framework?

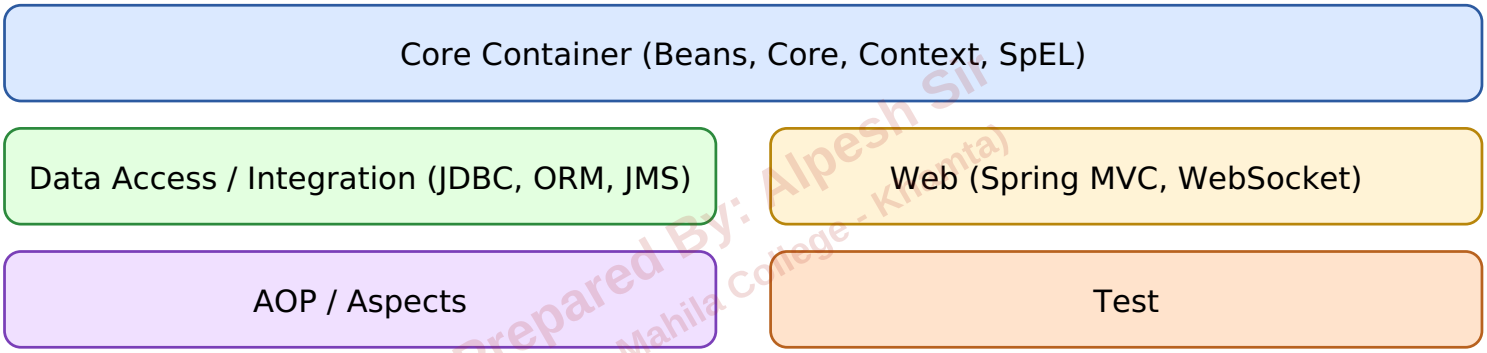
> **Spring** is a lightweight, open-source Java framework that provides comprehensive infrastructure support (DI, AOP, MVC, Data Access, Security) for building enterprise applications, reducing boilerplate code compared to plain J2EE/EJB.

Spring was created partly as a reaction against the complexity of early EJB, which required heavyweight configuration, special interfaces, and a full application server just to run simple business logic. Spring showed that the same enterprise-grade features — transaction management, security, remote access — could be achieved using plain ordinary Java objects, commonly called **POJOs (Plain Old Java Objects)**, configured externally rather than by forcing the business class itself to implement special framework interfaces. This made code easier to write, easier to unit-test outside a server, and easier to maintain — which is why Spring became the dominant framework for J2EE/Java EE development over the last two decades.

5.1.1 Importance and Benefits of Spring

- Loose coupling through Dependency Injection
- Lightweight — POJO-based programming model
- Modular — use only the parts you need (MVC, Data, Security...)
- Excellent integration with Hibernate, JDBC, JMS
- Testable code (POJOs are easy to unit test)

5.1.2 Spring Architecture Overview



5.2 Core Concepts of Spring

5.2.1 Dependency Injection (DI) and Inversion of Control (IoC)

> **IoC** is a principle where object creation and dependency management are handed over from the programmer to the Spring Container.
> **DI** is the technique used to implement IoC — the container "injects" required objects (dependencies) into a class instead of the class creating them itself.

In traditional programming, a class that needs another object (a "dependency") typically creates it directly using the `new` keyword — this is called **tight coupling**, because the class is now permanently bound to one specific implementation and cannot be easily swapped or tested with a fake/mock version. IoC "inverts" this control: instead of the class controlling how its dependencies are created, that control is handed over to an external container (the Spring IoC Container), which reads a configuration (XML, Java annotations, or Java config classes), creates all the required objects, and wires them together automatically. The practical benefit is huge for testing — during a unit test, a mock `Engine` object can be injected into `Car` instead of a real one, without changing a single line of the `Car` class.

Without DI (tight coupling):

```
class Car {
   Engine engine = new Engine();   // Car creates its own dependency
}
```

With DI (loose coupling):

```
class Car {
   private Engine engine;
   public Car(Engine engine) {   // Injected by Spring Container
       this.engine = engine;
```

```
    }
}
```

Types of DI: Constructor Injection (via constructor), Setter Injection (via setter methods).

5.2.1.1 Spring XML Configuration Example (classic approach)

Before annotation-based configuration became common, Spring beans and their dependencies were wired together using an external XML file — this remains important to understand because it makes the IoC concept very explicit and visual.

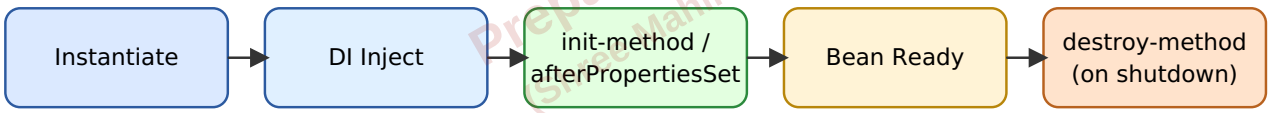
```
<!-- beans.xml -->
<beans>
  <bean id="engine" class="com.example.Engine"/>

  <bean id="car" class="com.example.Car">
     <constructor-arg ref="engine"/>   <!-- Constructor Injection -->
  </bean>
</beans>
```

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");
Car car = (Car) ctx.getBean("car");
```

Modern Spring/Spring Boot projects mostly replace this XML with annotations like `@Component`, `@Autowired`, and `@Configuration` classes — but the underlying IoC principle (container creates and wires the beans) stays exactly the same.

5.2.2 Bean Lifecycle in Spring

Instantiate → Populate Properties (DI) → setBeanName() → Aware interfaces → BeanPostProcessor(before) → afterPropertiesSet()/init-method → Bean Ready to Use → BeanPostProcessor(after) → destroy()/disposable



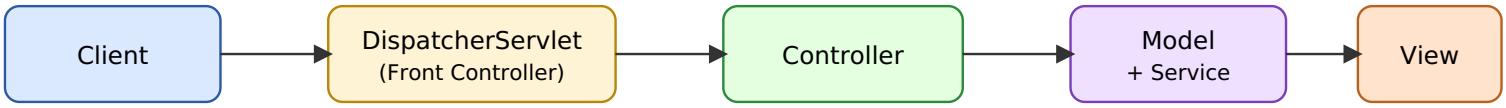
5.2.3 Spring ApplicationContext and BeanFactory

BeanFactory	ApplicationContext
Basic IoC container, lazy loading	Advanced container, extends BeanFactory
No support for AOP/i18n out-of-the-box	Supports AOP, event handling, internationalization
Rarely used directly now	Used in almost all real Spring applications

5.3 Spring MVC Basics

5.3.1 Spring MVC Architecture and Request Flow

Spring MVC is Spring's implementation of the classic MVC pattern (seen earlier in Unit 4), rebuilt around a single, central **Front Controller** instead of many independent servlets. Every single incoming request — no matter which URL it targets — first passes through the `DispatcherServlet`, which then looks up the correct `@Controller` method to handle it, based on URL mapping annotations like `@GetMapping`. Centralizing this routing logic in one place makes it far easier to apply common behavior (logging, security checks, exception handling) consistently across the entire application, instead of repeating that logic in every individual servlet.



All requests first reach the **DispatcherServlet** (Front Controller), which routes them to the right `@Controller`. The controller processes the request via a Service/Model, then returns a logical view name, which DispatcherServlet resolves and renders back to the client.

5.3.2 Simple Spring MVC Controller Example

```
@Controller
public class StudentController {

    @GetMapping("/student")
    public String showForm(Model model) {
        model.addAttribute("msg", "Welcome Student!");
        return "studentView";   // resolves to studentView.jsp
    }
}
```

5.4 Aspect-Oriented Programming (AOP) in Spring

AOP lets you add cross-cutting concerns (logging, security, transactions) to your code **without modifying the business logic itself**, by defining them separately as "Aspects."

Some concerns, like logging every method call, checking security permissions, or wrapping a method in a database transaction, are needed across *many unrelated* classes throughout an application — this is why they are called "cross-cutting" concerns; they cut across the normal class-by-class structure of the program. Without AOP, a developer would have to copy-paste the same logging/security code into every single business method, cluttering the code and making it hard to maintain (if the logging format needs to change, it must be changed in hundreds of places). AOP solves this by letting the developer write that shared logic exactly once, in a separate "Aspect" class, and then declaratively specify (via a Pointcut expression) which methods it should automatically apply to — Spring then weaves this logic in at runtime, invisibly, without the business class needing any awareness of it.

Term	Meaning
Aspect	A module containing cross-cutting logic (e.g. LoggingAspect)
Join Point	A point during execution (e.g. a method call) where an aspect can be applied
Pointcut	An expression that selects which join points to apply advice to
Advice	The actual action taken (Before, After, Around a join point)

Simple Logging Aspect:

```
@Aspect
@Component
public class LoggingAspect {

    @Before("execution(* com.example.service.*.*(..))")
    public void logBefore() {
        System.out.println("Method execution started...");
    }
}
```

5.5 Spring Boot Fundamentals

Spring Boot is built on top of Spring Framework to eliminate boilerplate configuration. It provides **auto-configuration**, an embedded

server, and "starter" dependencies so a full application can run with almost zero XML setup.

Classic Spring applications, while powerful, still required a fair amount of manual XML or Java configuration to wire together components like a DataSource, a ViewResolver, or a Servlet Container — and the developer had to separately install and configure Tomcat before deployment. Spring Boot was created to remove this friction entirely: it examines which libraries are present on the classpath (e.g. if it sees a database driver JAR, it auto-configures a DataSource with sensible defaults) and bundles an embedded Tomcat server directly inside the application itself, so the whole project can be run as a single executable JAR file with just one command — no separate server installation needed. This is why Spring Boot has become the default starting point for almost all new Java web/enterprise projects today.

5.5.1 Features and Advantages

- Auto-Configuration — sensible defaults, less manual setup
- Embedded servers (Tomcat/Jetty) — no separate server installation needed
- Starter Dependencies — one dependency pulls in everything needed (e.g. spring-boot-starter-web)
- Production-ready features (Actuator for monitoring)

5.5.2 Architecture of Spring Boot

5.5.3 Important Spring Boot Annotations

Annotation	Purpose
`@SpringBootApplication`	Combines `@Configuration` + `@EnableAutoConfiguration` + `@ComponentScan`; marks the main class.
`@RestController`	Combines `@Controller` + `@ResponseBody`; returns data (JSON) directly, not a view.
`@RequestMapping`	Maps a URL/HTTP method to a controller method or class.


```
@SpringBootApplication
public class MyApp {
    public static void main(String[] args) {
        SpringApplication.run(MyApp.class, args);
    }
}

@RestController
@RequestMapping("/api")
class StudentApi {
    @GetMapping("/hello")
    public String hello() {
        return "Hello from Spring Boot!";
    }
}
```

5.6 Spring Boot Data Access

Recall from Unit 1 that plain JDBC requires seven repetitive steps (load driver, open connection, create statement, execute, process ResultSet, close connection, handle exceptions) for every single database operation. Spring's data-access support exists to eliminate this repetition at two different levels of abstraction: `JdbcTemplate` removes the boilerplate around plain SQL, while **Spring Data JPA** goes a step further and removes the need to write SQL or DAO implementation code at all for standard CRUD operations.

5.6.1 Spring Boot JDBC — JdbcTemplate

`JdbcTemplate` removes boilerplate JDBC code (no manual Connection/Statement/ResultSet handling, no try-catch for every query).

```
@Autowired
JdbcTemplate jdbcTemplate;

public List<Student> getAll() {
    return jdbcTemplate.query("SELECT * FROM student",
        (rs, rowNum) -> new Student(rs.getInt("id"), rs.getString("name")));
}
```

5.6.2 Spring Boot Data JPA

Spring Data JPA works on top of Hibernate/JPA and lets you perform CRUD operations by simply **extending an interface** — no implementation code needed.

```
public interface StudentRepository extends JpaRepository<Student, Integer> {
    // findAll(), save(), deleteById(), findById() available automatically!
    List<Student> findByName(String name);   // custom query by method name
}
```

5.6.3 Complete Layered Spring Boot REST CRUD Example

This full example ties together everything learned in Unit 5 — Entity, Repository, Service, and Controller layers — mirroring the N-Tier architecture introduced in Unit 1.


```
// Student.java (Entity / Model)
@Entity
public class Student {
    @Id @GeneratedValue
    private int id;
    private String name;
    private int marks;
    // getters and setters
}

// StudentRepository.java (Data Access Layer)
public interface StudentRepository extends JpaRepository<Student, Integer> { }

// StudentService.java (Business Layer)
@Service
public class StudentService {
    @Autowired
    private StudentRepository repo;

    public List<Student> getAllStudents() { return repo.findAll(); }
    public Student addStudent(Student s)   { return repo.save(s); }
    public void deleteStudent(int id)      { repo.deleteById(id); }
}

// StudentController.java (Presentation / API Layer)
@RestController
@RequestMapping("/api/students")
public class StudentController {
    @Autowired
    private StudentService service;

    @GetMapping
    public List<Student> getAll() { return service.getAllStudents(); }
```

```
    @PostMapping
    public Student add(@RequestBody Student s) { return service.addStudent(s); }

    @DeleteMapping("/{id}")
    public void delete(@PathVariable int id) { service.deleteStudent(id); }
}
```

Notice how closely this maps to the N-Tier diagram from Unit 1: the Controller only handles HTTP routing, the Service holds business rules (here minimal, but this is where validations/calculations would go), and the Repository (DAO layer) is the only place that touches the database — each layer can be modified independently without breaking the others.

5.7 Exception Handling in Spring Boot MVC

Annotation	Use
`@ExceptionHandler`	Handles a specific exception thrown inside one controller
`@ControllerAdvice`	Global exception handler applied across all controllers

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(StudentNotFoundException.class)
    public ResponseEntity<String> handleNotFound(StudentNotFoundException ex) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT_FOUND);
    }
}
```

Exam Important Questions & Answers

Section A — 2/3 Marks Questions

Q1. What is J2EE?

A: J2EE (Java EE) is a platform built over Core Java, providing APIs (Servlet, JSP, EJB, etc.) and a runtime container to build large-scale, multi-tier enterprise/web applications.

Q2. Differentiate two-tier and three-tier architecture.

A: Two-tier has only Client and Database (logic mixed with client). Three-tier separates Presentation, Business Logic, and Data into independent layers, improving maintainability and scalability.

Q3. What is JDBC? List its types of drivers.

A: JDBC is a Java API to connect and operate on relational databases. Four driver types: Type 1 (JDBC-ODBC Bridge), Type 2 (Native-API), Type 3 (Network Protocol), Type 4 (Thin Driver — most used).

Q4. Differentiate Statement and PreparedStatement.

A: Statement executes static SQL and is compiled every time; PreparedStatement is precompiled, supports parameters (?), is faster on repeated execution, and prevents SQL injection.

Q5. What is a Servlet? Explain its life cycle.

A: A Servlet is a Java class that handles client requests on the server. Life cycle: init() (once, setup) → service() (per request, calls doGet/doPost) → destroy() (once, cleanup).

Q6. What is session tracking? Name its techniques.

A: Since HTTP is stateless, session tracking maintains user state across requests. Techniques: URL Rewriting, Hidden Form Fields, Cookies, and the Session API (HttpSession).

Q7. Differentiate JSP and Servlet.

A: Servlet = Java code with embedded HTML (good for logic); JSP = HTML with embedded Java (good for UI), auto-converted into a servlet by the container.

Q8. What are JSP implicit objects? Name any four.

A: Objects automatically available in JSP without declaration: request, response, session, application, out, pageContext.

Q9. Differentiate include action and forward action in JSP.

A: include action merges the response and returns control to the original page; forward action passes control permanently to another resource and clears the buffer.

Q10. What is EJB? Name its types.

A: EJB (Enterprise Java Bean) is a managed server-side component for business logic. Types: Session Bean (Stateless/Stateful), Entity Bean, Message-Driven Bean.

Q11. What is MVC architecture? Explain its components.

A: MVC separates an app into Model (data/logic), View (UI/JSP), and Controller (Servlet — receives requests, coordinates Model and View).

Q12. What is Hibernate? List its features.

A: Hibernate is an ORM framework mapping Java classes to DB tables. Features: open source, database-independent, automatic table mapping, caching, HQL support.

Q13. What is HQL? How is it different from SQL?

A: HQL (Hibernate Query Language) is object-oriented — it queries Java class names/properties instead of table/column names, and is database-independent, unlike SQL.

Q14. What is Dependency Injection? What are its types?

A: DI is a technique where the Spring Container supplies (injects) an object's dependencies instead of the object creating them itself, achieving loose coupling. Types: Constructor Injection, Setter Injection.

Q15. Differentiate BeanFactory and ApplicationContext.

A: BeanFactory is the basic IoC container with lazy loading; ApplicationContext extends it, adding AOP, event handling, and internationalization support — used in almost all real applications.

Q16. What is AOP? Define Aspect, Join Point, Pointcut, Advice.

A: AOP adds cross-cutting concerns (logging, security) without touching business logic. Aspect = module of cross-cutting code; Join Point = point of execution (e.g. method call); Pointcut = expression selecting join points; Advice = action taken (Before/After/Around).

Q17. What is Spring Boot? How is it different from Spring Framework?

A: Spring Boot is built on Spring Framework to remove manual configuration via auto-configuration, embedded servers, and starter dependencies — enabling rapid application development with minimal setup.

Q18. What is the use of @ControllerAdvice?

A: @ControllerAdvice defines a global, centralized exception handler that applies across all controllers in a Spring Boot application, unlike @ExceptionHandler which is local to one controller.

Section B — Long/Descriptive Questions (5–7 Marks)

Q1. Explain the three-tier architecture of J2EE with a diagram.

A: (See Unit 1, Section 1.2.2) — Explain Presentation, Business, and Data tiers, their responsibilities, and advantages: independent scalability, security, maintainability, and reusability of the business layer across multiple clients (web, mobile).

Q2. Explain the JDBC architecture and the steps to create a JDBC application with an example.

A: (See Unit 1, Sections 1.5.1 & 1.5.4) — Explain the 7 steps: import packages, load driver, create connection, create statement, execute query, process ResultSet, close connection. Give a full code example with a SELECT query.

Q3. Explain the Servlet life cycle in detail with a diagram.

A: (See Unit 2, Section 2.4) — Explain loading, init() (once), service() (per request, dispatches doGet/doPost), destroy() (once). Mention that the container manages threading — one servlet instance handles multiple requests using multithreading.

Q4. Explain all types of session tracking techniques with examples.

A: (See Unit 2, Section 2.9) — Explain URL Rewriting, Hidden Form Fields, Cookies, and HttpSession API with code snippets and their pros/cons.

Q5. Explain JSP life cycle and its elements (Directive, Scripting, Action) with examples.

A: (See Unit 3, Sections 3.3 & 3.4) — Explain translation → compilation → execution, then jspInit()/_jspService()/jspDestroy(), followed by directive/scripting/action element tables with syntax examples.

Q6. Explain MVC architecture and how it is implemented using Servlet, JSP, and JavaBean.

A: (See Unit 4, Section 4.2) — Explain Model (JavaBean/business logic), View (JSP for UI), Controller (Servlet handling request routing), and draw the request flow diagram.

Q7. Explain Hibernate architecture along with configuration and mapping files.

A: (See Unit 4, Section 4.3) — Explain SessionFactory, Session, Transaction, hibernate.cfg.xml, .hbm.xml/annotations, and a simple save example using HQL.

Q8. Explain Dependency Injection and IoC container in Spring with an example.

A: (See Unit 5, Section 5.2.1) — Explain IoC principle, DI implementation, constructor vs setter injection with tight-coupling vs loose-coupling code comparison.

Q9. Explain Spring MVC architecture and request flow with DispatcherServlet.

A: (See Unit 5, Section 5.3.1) — Explain DispatcherServlet as Front Controller, routing to @Controller, Model population, and View resolution, with a diagram and code example.

Q10. Explain Spring Boot features and important annotations with a simple REST example.

A: (See Unit 5, Section 5.5) — Explain auto-configuration, embedded server, starters, @SpringBootApplication, @RestController, @RequestMapping with a runnable code example.

Summary — Whole Syllabus at a Glance

Unit	Core Topic	Key Takeaway (One Line)
1	J2EE & JDBC	J2EE = enterprise platform over Core Java; JDBC connects Java apps to databases using Driver → Connection → Statement → ResultSet.
2	Servlet	Server-side Java class handling HTTP requests via init() → service() → destroy(); session tracking maintains state over stateless HTTP.
3	JSP	HTML + Java mixed page, auto-converted to a servlet; uses directives, scriptlets, actions, implicit objects, and EL/JSTL for clean UI code.
4	EJB, MVC, Hibernate	EJB = managed business components; MVC separates data/UI/control; Hibernate = ORM mapping Java objects to DB tables via HQL.
5	Spring & Spring Boot	Spring uses DI/IoC and AOP for loosely-coupled apps; Spring Boot auto-configures everything with starters and embedded servers for rapid development.

One-Line Memory Chain for the Whole Course:
Client Request → Servlet (Controller) → JSP (View) → Business Logic/EJB (Model) → Hibernate (ORM) → Database
...all made easier today by → Spring (DI+AOP+MVC) → Spring Boot (Auto-config + Starters + Embedded Server)

Quick Revision Checklist Before Exam:

- Draw: Three-Tier Architecture, JDBC Architecture, Servlet Life Cycle, JSP Life Cycle, MVC Diagram, Hibernate Architecture, Spring MVC Flow.
- Write: 7 Steps of JDBC, Servlet Life Cycle methods, JSP implicit objects, EJB types, Spring annotations.
- Compare: Statement vs PreparedStatement, Servlet vs JSP, include vs forward, BeanFactory vs ApplicationContext.