Model 1 Architecture

- **Technologies: Servlet + JSP.**
- **How it works: Each Servlet/JSP handles both the presentation and business logic. Servlet may handle some logic, but JSP can also directly interact with the database or backend.**
- **Advantages:**
  - **Simple and quick to develop.**
  - **Suitable for small applications or prototypes.**
- **Disadvantages:**
  - **Decentralized navigation: Each JSP may contain logic to decide the next page → maintenance headache.**
  - **Mixing of concerns: Business logic, presentation, and data access are often mixed → hard to test.**
  - **Hard to scale: Modifying a JSP may require recompiling/deploying multiple pages.**

Model 2 Architecture (MVC)

- **Based on: Model-View-Controller design pattern.**
- **Components:**
  - Model: Business logic and state of the application (JavaBeans, POJOs, or service classes).
  - View: JSPs, Thymeleaf, or frontend templates (responsible for rendering UI).
  - Controller: Servlet or filter acting as the Front Controller → intercepts all requests and dispatches to the appropriate model and view.
- **Advantages:**
  - Centralized navigation: Only the controller handles routing → easier to maintain.
  - Separation of concerns: Model, View, and Controller are independent → better testing and reuse.
  - Extensible and maintainable: Easier to scale for large applications.

- **Disadvantages:**
    - Initial setup of a centralized dispatcher can be tedious.
    - More code compared to Model 1 for small apps.
    - Requires understanding of MVC flow.

Spring MVC is essentially a Model 2 implementation that provides ready-made components like DispatcherServlet (Front Controller), HandlerMapping, and ViewResolver to simplify development.

What is MVC?

MVC (Model–View–Controller) is a **design pattern** for web (and desktop) applications that ensures **separation of concerns**:

- **Model** → Business/Data logic
  (JavaBeans, Service classes, DAO, POJOs, Entities)
- **View** → UI/Presentation logic
  (JSP, Thymeleaf, React, Angular, Vue , Next JS etc.)
- **Controller** → Request handling & navigation
  (Servlets, Filters, Spring MVC controllers, Struts actions)

Front Controller Pattern

- Additional design pattern
- Adds a **single entry point** for *all requests*.
- Acts like a **gatekeeper / traffic controller**.
- Common tasks:
    - Authentication
    - Logging
    - Request pre-processing
    - Dispatching to the right controller/action

Examples:

- **Servlet world:** A DispatcherServlet in Spring MVC, or a custom FrontControllerServlet.
- **Struts 2:** Uses FilterDispatcher (Filter-based Front Controller).

MVC Flow (with Front Controller) – Login Example

1. Client sends request:
   POST http://localhost:8080/web_mvc/login with request body - email and password.
2. Front Controller intercepts:
   The FrontController servlet receives every request. Based on an action (e.g., action=login), it either:
   - Forwards to a dedicated UserController (if you maintain a separate controller layer), or
   - Directly calls the appropriate method in UserService (simpler approach).

The FrontController centralizes navigation and ensures consistent request handling.

3. Service Layer (optional but recommended):
   UserService handles business logic, like validating credentials. It may store conversational state if needed (but usually request/session scope is enough).
4. DAO Layer Access:
   UserService calls UserDAO to query the database and get user information.
5. Return to Front Controller:
   UserService returns results (e.g., a User object or a success/failure flag). The servlet stores this in request/session attributes using request.setAttribute() or session.setAttribute().
6. Forward to JSP (View Layer):
   The FrontController then forwards the request to a JSP, e.g., login-result.jsp. The JSP reads the attributes and generates the HTML response.
7. Response Back to Client:
   The JSP sends the HTML to the client. Control returns through the servlet, completing the request-response cycle.