2.1 Introduction to Spring Boot, MVC Architecture, Tomcat and Dispatcher Servlet

🌐 ***Client-Server Architecture*** **–**

Client-Server Architecture is a **model** where ***two*** **types of devices** interact with each other:

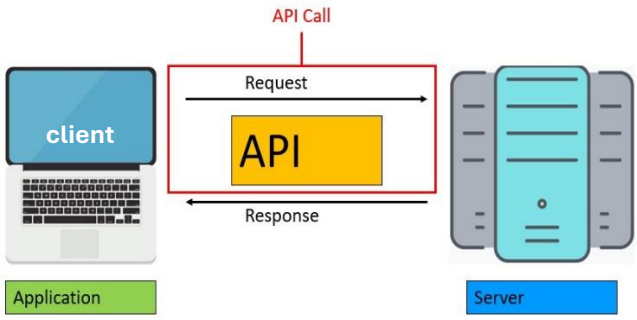
Components:

Client: The *user-side* (browser, app, device) that sends ***requests*** to access data or services.

Server: The *provider-side* that **processes the request**, runs the logic, accesses the database, and sends ***responses*** back.

How It Works:

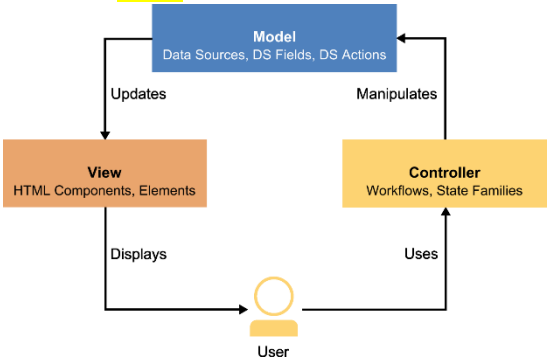
1. The ***client*** sends a ***request*** (e.g., fetch user data).
2. The ***server*** receives the **request**, performs operations (e.g., database query).
3. The ***server*** sends a ***response*** (e.g., user data in **JSON**).
4. The ***client*** displays the data or continues further action.



📡 **Role of API in Client-Server Architecture:**

API (Application Programming Interface) is the ***middleman*** that allows the ***client*** and ***server*** to ***communicate***.

What is MVC Architecture?

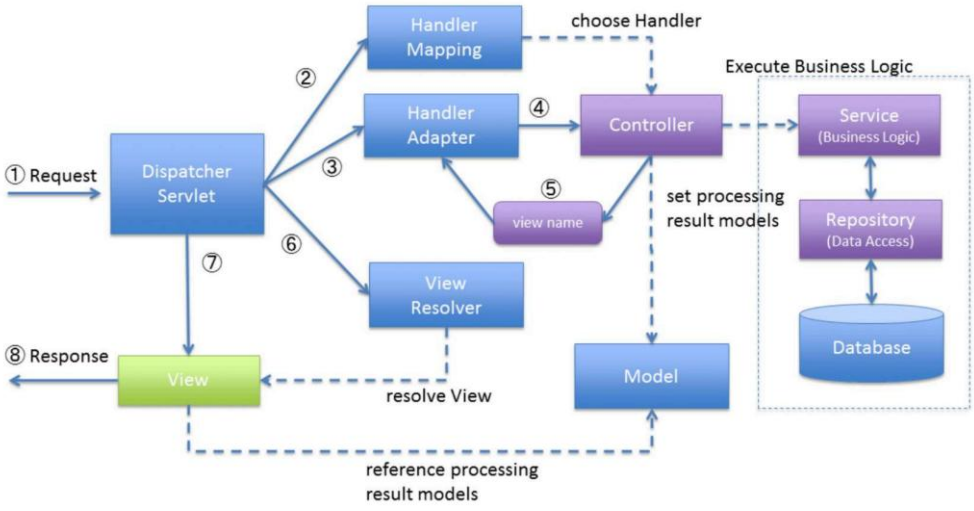


MVC stands for ***Model–View–Controller***, a ***design pattern*** used to build **organized** and **scalable applications** — especially web applications.

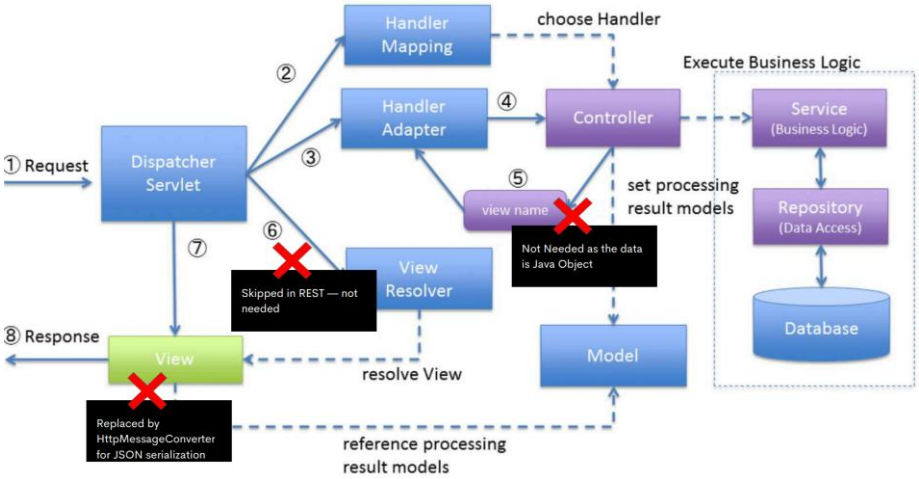
Components of MVC:

Component	Role
Model	🧠 Handles the ***data*** and ***business logic*** (e.g., database access, calculations).
View	🎯 Handles the ***UI*** – what the user sees (**HTML**, frontend templates, etc).
Controller	🎮 Handles ***user input***, **updates** the **Model**, and **decides** which **View** to show.

Spring MVC (Traditional Flow):



Spring MVC with REST APIs:



How does a Web Server works in Spring Boot?

Browser/Postman
↓
Embedded Tomcat
↓
DispatcherServlet
↓
HandlerMapping → which controller?
↓
HandlerAdapter → call controller method
↓
Controller method returns **POJO/DTO**
↓
HttpMessageConverter (e.g., Jackson → JSON)
↓
DispatcherServlet writes to HttpServletResponse
↓
Client gets **JSON**

🌐 **What is a *REST API* ?**

- *REST API* stands for *Representational State Transfer Application Programming Interface*.
- It's a *way* for *systems* to *communicate* over *HTTP* using standard operations like *GET*, *POST*, *PUT*, and *DELETE*.

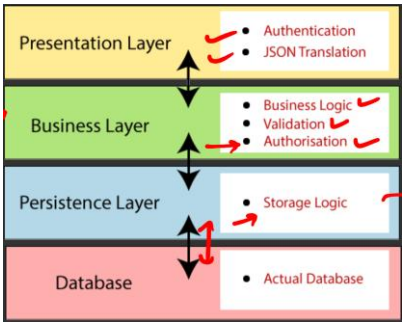
Example:

- *GET /users:* Retrieve a list of all users.
- *GET /users/{id}:* Retrieve a specific user by ID.
- *POST /users:* Create a new user.
- *PUT /users/{id}:* Update an existing user by ID.
- *PATCH /users/{id}:* Partially update an existing user by ID.
- *DELETE /users/{id}:* Delete a user by ID.

Method	Purpose
GET	Read/fetch data
POST	Create new resource
PUT	Update whole resource
PATCH	Update part of resource
DELETE	Delete resource

Layered Architecture:

- Layered Architecture is a ***software design pattern*** where an application is organized into ***separate layers***, and each layer has a specific role.
- The goal is to ***separate concerns***, ***improve modularity***, and make the system ***easier to maintain*** and test.



@Sibasundarj8

1. *Presentation Layer:*
   - Also called the *User Interface Layer* or *Web Layer*.
   - It's the *first point of contact* between the **user** and the **system**.
   - *Responsibilities:*
     - Receives *HTTP* requests
     - Translates *JSON/XML* input into *Java objects*
     - Performs *authentication*
     - Sends *HTTP* responses (usually in *JSON*)
   - Handled by: *Controllers* (*@RestController* in Spring Boot)

2. *Business Layer:*
   - Also known as the *Service Layer*.
   - Contains the *core business logic* of the application.
   - **Responsibilities:**
     - Enforces *business rules*
     - Performs data *validation*
     - Handles *authorization* (e.g., who can do what)
   - Handled by: *Services* (*@Service* classes in Spring Boot)

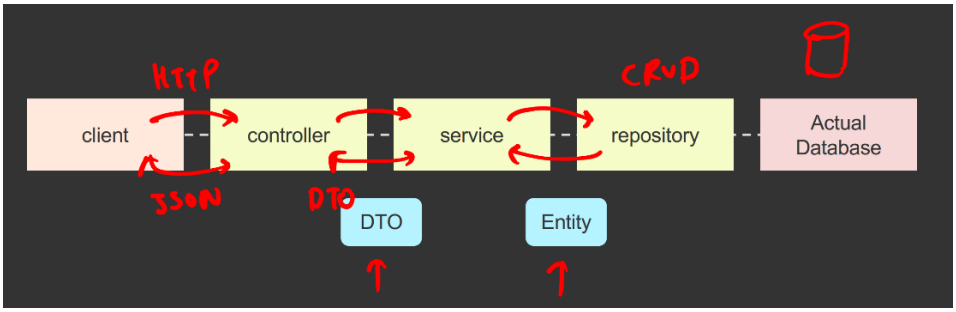
3. *Persistence Layer:*
   - Also called the *Data Access Layer* (DAO/Repository).
   - Talks to the *database* using *JPA/Hibernate* or *JDBC*.
   - **Responsibilities:**
     - Executes *CRUD operations*
     - Manages database *transactions*
     - **Maps Java objects** (**Entities**) to *database tables*
   - Handled by: *Repository* **Interfaces** (*@Repository* in Spring Boot)

4. *Database Layer:*
   - The *physical storage* system.
   - Usually a **relational database** like *MySQL*, *PostgreSQL*, *Oracle*, etc.
   - **Responsibilities:**
     - *Stores* and *retrieves* persistent *data*

Interacts only with the *Persistence Layer*

Spring Boot Web Project Structure:

1. ***Client***
   - This is the *frontend* or external system (like **browser**, **Postman**, or mobile app).
   - It communicates with the **backend** via *HTTP*.
   - Sends requests as *JSON*, receives responses as *JSON* too.



2. ***Controller Layer:***
   - Acts as the *entry point* to the *backend*.
   - Handles *incoming HTTP* requests, maps them to functions using annotations like *@GetMapping*, *@PostMapping*, etc.
   - Converts *JSON* ↔ *DTO* (Data Transfer Object).

Talks to the **Service layer** to perform business logic.

3. ***Service Layer***
   - Contains **business logic** of the application.
   - Receives *DTO* from the *Controller*, *converts* it into *Entity* (if needed).
   - Calls the Repository to perform *CRUD operations* on the database.
   - Returns *results* (often wrapped back into DTOs) to the controller.

4. ***Repository Layer:***
   - **Directly** interacts with the **database** using *JPA* or *Hibernate*.
   - Handles *CRUD operations*: **Create**, **Read**, **Update**, **Delete**.
   - Works with *Entity* **classes** that represent database tables.

5. ***Actual Database:***
   - Stores *persistent data*.
   - The final destination for **storing** or **retrieving** application data.

📦 ***DTO (Data Transfer Object):***
- A **lightweight object** used to *transfer data* between *client* ↔ *controller* and *controller* ↔ *service*.
- Keeps the **API** *decoupled* from the internal *Entity structure*.

🏛️ ***Entity:***
- A class annotated with *@Entity* that maps to a table in the database.

Used by the *Repository* layer to perform operations.

@Sibasundarj8