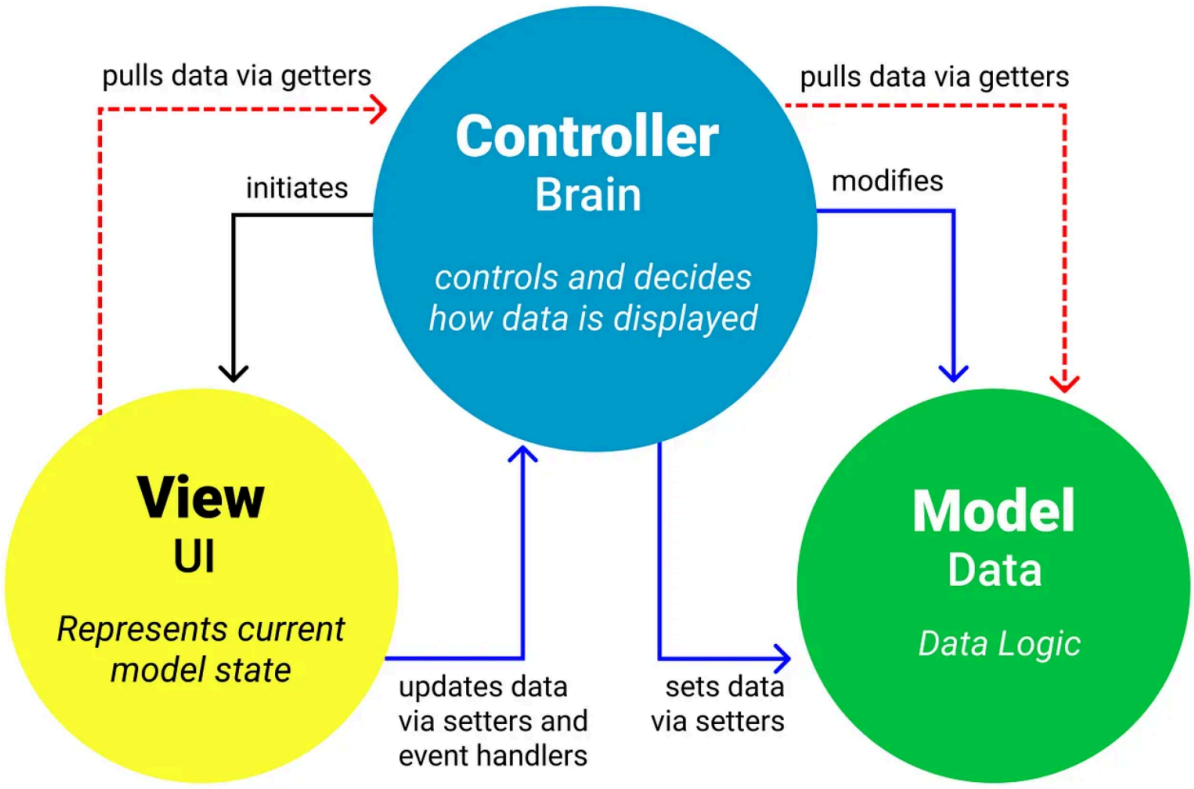
Web application: Deploy them on web

MVC Design pattern:



MVC is a software design pattern that separates an application into three interconnected components:

Model:

- Represents the data and business logic of the application.

- Manages the data, responds to queries, and updates the Controller about any changes.

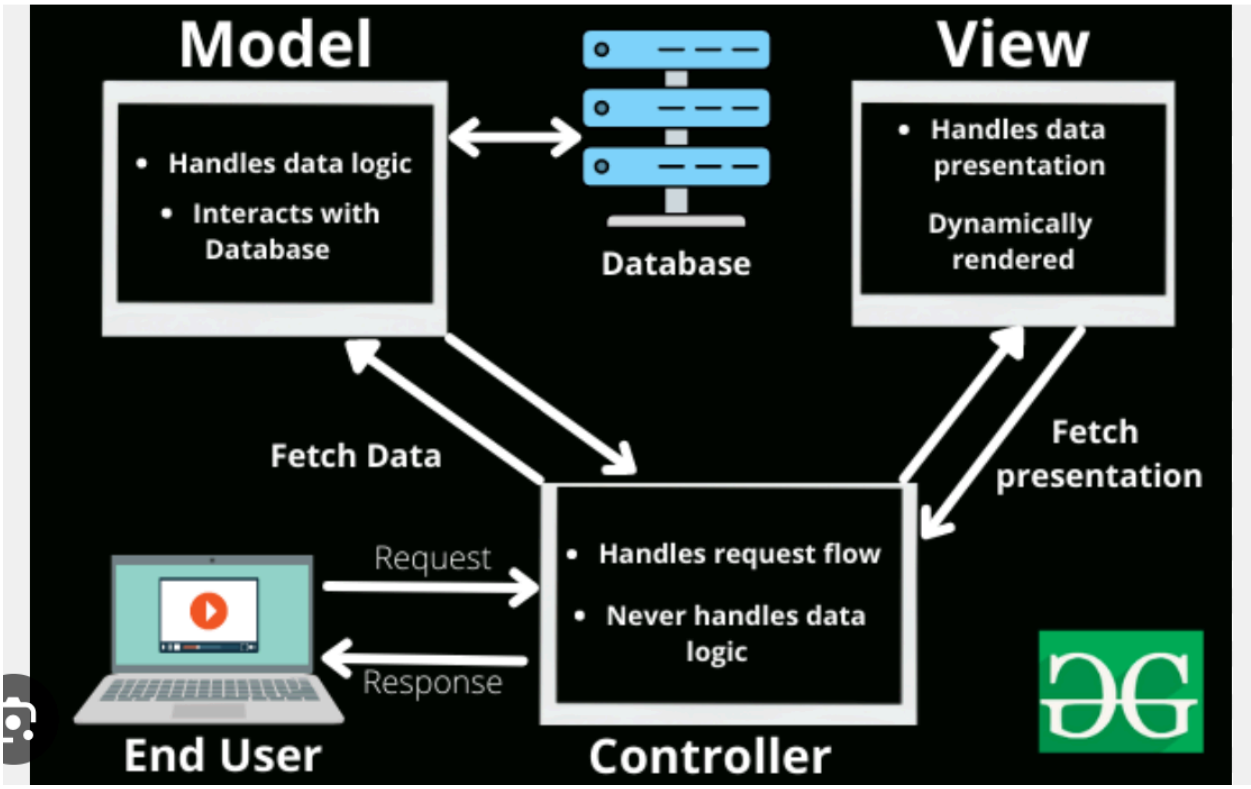
View:

- Presents the data to the user and handles user interface interactions.
- Receives input from users and sends it to the Controller for processing.

Controller:

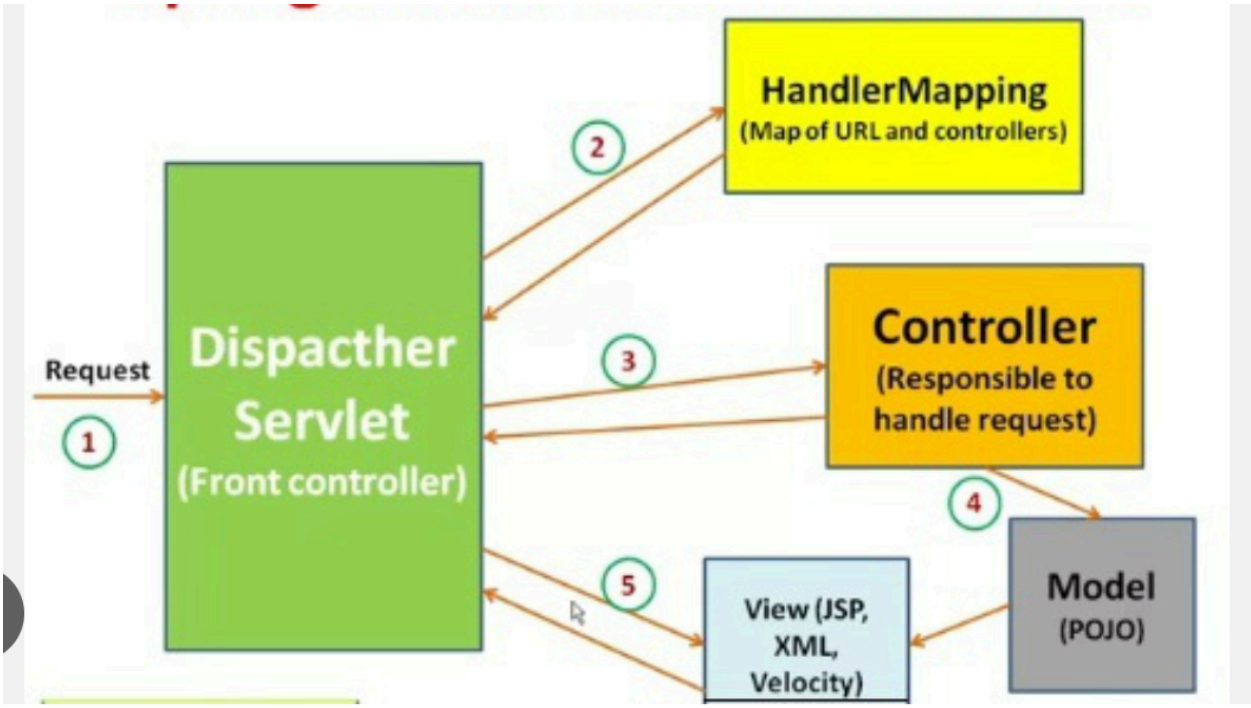
- Acts as an intermediary between the Model and View.
- Receives user input from the View, processes it (updates the Model if necessary), and updates the View.

The separation of concerns provided by MVC promotes modularity, making it easier to develop, test, and maintain software.

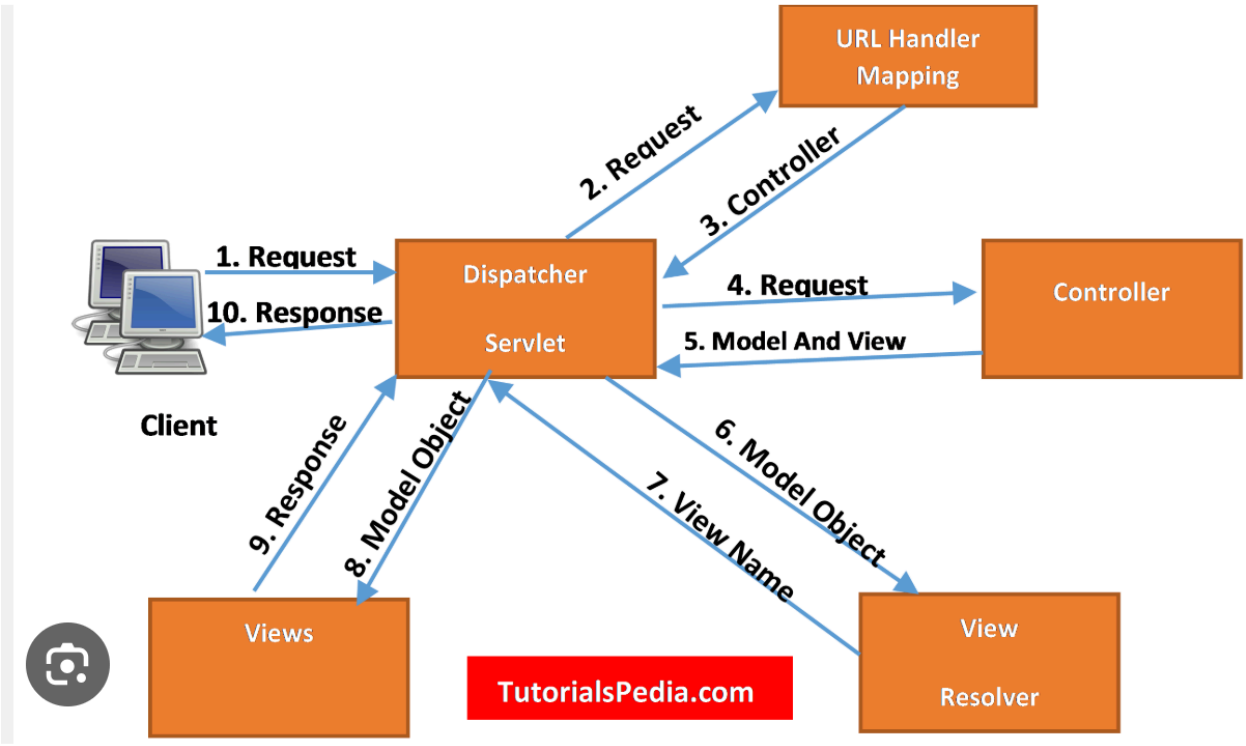


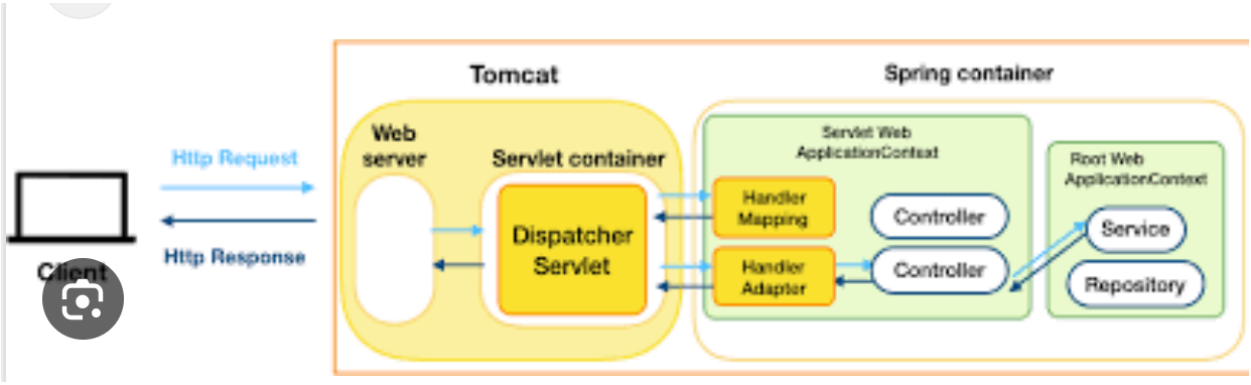
Front Controller:

In Spring MVC, the Front Controller is a design pattern implemented by the DispatcherServlet. It serves as the central entry point for all incoming web requests in a Spring-based web application.
Here's a breakdown of its role:
Single Entry Point:
Instead of having multiple entry points for different types of requests, the DispatcherServlet acts as the single recipient for all requests, centralizing the request handling process.
Request Delegation:
Upon receiving a request, the DispatcherServlet is responsible for determining which specific controller and handler method should process that request. This decision is made based on configured HandlerMapping implementations, often utilizing annotations like @RequestMapping.
Workflow Management:
It orchestrates the entire request processing workflow, including:



Handler Mapping: Identifying the appropriate controller and method.
Handler Execution: Invoking the chosen handler method.
View Resolution: If the handler returns a view name, the DispatcherServlet consults a ViewResolver to locate and render the correct view (e.g., JSP, Thymeleaf template).
Response Generation: Preparing the final response to be sent back to the client.
Essentially, the DispatcherServlet streamlines the request handling process, providing a flexible and extensible architecture for building web applications in Spring MVC.





1 Core Stereotype Annotations (marking Spring-managed beans): Spring will go and create the object for the classes annotated with these annotations

Annotation	Where Used	Purpose
`@Controller`	On a class	Marks a class as a Spring MVC controller (returns view names by default).

`@RestController`	On a class	Shortcut for `@Controller` + `@ResponseBody` — the return value is written directly to the HTTP response (good for REST APIs).
`@Service`	On a class	Marks a service-layer bean (business logic).
`@Repository`	On a class	Marks a data-access-layer bean; also helps Spring translate persistence exceptions.
`@Component`	On a class	Generic Spring-managed bean (no special role).

All these are detected if you enable component scanning with `@ComponentScan`

2️⃣ Configuration & Bootstrapping Annotations

Annotation	Where Used	Purpose
`@Configuration`	On a class	Indicates the class contains Spring bean definitions (@Bean methods).
`@Bean`	On a method	Declares a bean in a @Configuration class.
`@ComponentScan`	On a class	Tells Spring which packages to scan for annotated components (@Controller, @Service, etc.).
`@EnableWebMvc`	On a class	Turns on Spring MVC — registers DispatcherServlet, handler mappings, message converters, etc.
`@Import`	On a class	Imports another @Configuration class.

3️⃣ Request Mapping & Handling Annotations

Annotation	Where Used	Purpose
@RequestMapping	On a class/method	Maps HTTP requests to handler methods (supports path, HTTP method, params, headers).
@GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @PatchMapping	On a method	Shortcuts for @RequestMapping(method = RequestMethod.GET/POST/...).
@PathVariable	On a method parameter	Binds a URI template variable to a method parameter.
@RequestParam	On a method parameter	Binds a query parameter or form field to a method parameter.
@RequestBody	On a method parameter	Binds the request body to a method parameter (e.g., JSON to object).
@ResponseBody	On a method or class	Writes method return value directly to the HTTP response body (JSON, XML, text).
@ModelAttribute	On a parameter or method	Binds form data to a model object, and/or adds attributes to the model.
@RequestHeader	On a method parameter	Binds an HTTP header value to a parameter.
@CookieValue	On a method parameter	Binds a cookie value to a parameter.

4️⃣ Data Binding & Validation

Annotation	Where Used	Purpose

`@Valid / @Validated`	On a parameter	Triggers bean validation on method arguments.
`@InitBinder`	On a method	Customizes data binding for request parameters.

5️⃣ Example: Putting them together

```
@Controller
@RequestMapping("/users")
public class UserController {

    @GetMapping("/{id}")
    @ResponseBody
    public User getUser(@PathVariable int id) {
        // fetch user by id
        return new User(id, "Alice");
    }

    @PostMapping
    public String addUser(@ModelAttribute User user) {
        // save user
        return "redirect:/users/" + user.getId();
    }
}
```

Here's what happens:

- `@Controller` → Marks the class as a Spring MVC controller
- `@RequestMapping("/users")` → All methods in this controller handle `/users/...`
- `@GetMapping("/{id}")` → GET `/users/123` calls `getUser()`
- `@PathVariable int id` → Extracts 123 from the URL

- `@ResponseBody` → Writes returned `User` object directly as JSON
- `@PostMapping` → Handles POST `/users`
- `@ModelAttribute User user` → Binds form fields to a `User` object

RequestMappingHandlerMapping

1️⃣ Where it fits in the Spring MVC flow

When an HTTP request hits your app:

1. `DispatcherServlet` receives it.
2. It asks a HandlerMapping to find *which handler* (controller method) should process this request.
3. `RequestMappingHandlerMapping` is one such HandlerMapping — the one that understands `@RequestMapping`, `@GetMapping`, etc.
4. Once it finds the matching method, it returns a HandlerMethod (method + controller instance).
5. The HandlerAdapter then calls that method with the right arguments.

2️⃣ What `RequestMappingHandlerMapping` does

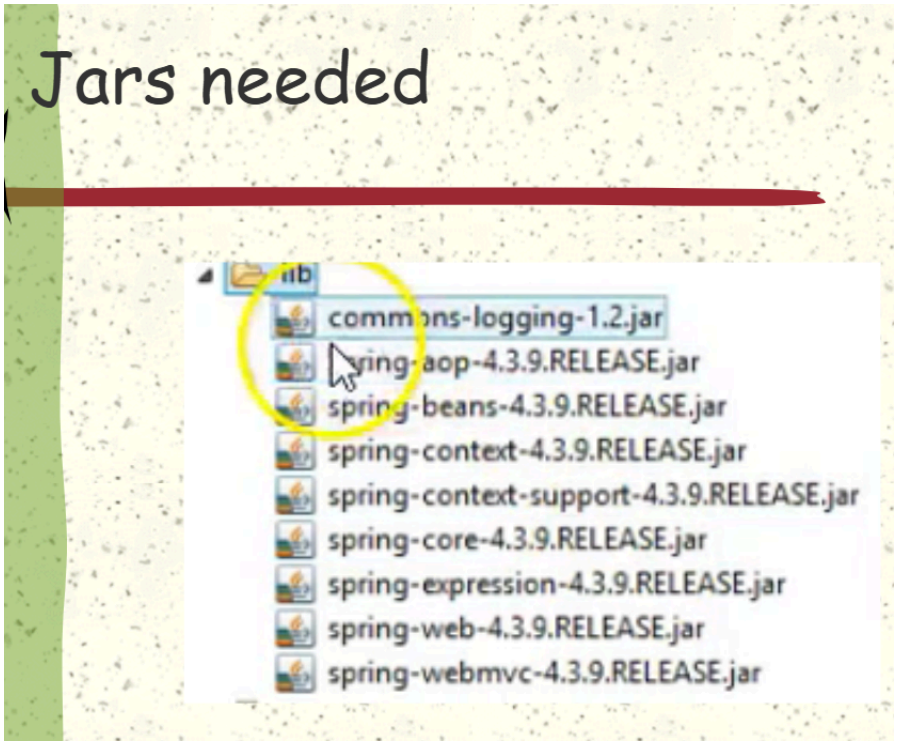
- At application startup, it scans all beans in the application context.
- It looks for:
    - Classes annotated with `@Controller` or `@RestController`
    - Methods annotated with `@RequestMapping` or shortcuts like `@GetMapping`, `@PostMapping`, etc.

For each such method, it builds a mapping table:

```
[HTTP method, path pattern]  →  HandlerMethod
```

-
- At runtime, when a request comes in, it looks up that table and returns the matching method.

Spring mvc project (no spring boot):



1️⃣ Add dependencies (Maven example)

Since you're not using Spring Boot, you must explicitly add:

- Spring MVC jars
- Embedded Tomcat jars
- Servlet API

<dependencies>

```
    <!-- Spring MVC -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>6.1.6</version> <!-- use a version matching your setup -->
    </dependency>

    <!-- Embedded Tomcat, no need of external server -->
    <dependency>
        <groupId>org.apache.tomcat.embed</groupId>
        <artifactId>tomcat-embed-core</artifactId>
        <version>10.1.24</version>
    </dependency>
    <dependency>
        <groupId>org.apache.tomcat.embed</groupId>
        <artifactId>tomcat-embed-jasper</artifactId> <!-- for JSP support -->
        <version>10.1.24</version>
    </dependency>

    <!-- Servlet API -->
    <dependency>
        <groupId>jakarta.servlet</groupId>
        <artifactId>jakarta.servlet-api</artifactId>
        <version>6.0.0</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

2️⃣ Create the Spring MVC Configuration

Create a `@Configuration` class:

```
package com.example.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.example")
public class AppConfig {
}
```

3️⃣ Create the Spring `DispatcherServlet` Initializer

This replaces `web.xml` for annotation-based configuration:

```
package com.example.init;

import com.example.config.AppConfig;
import
org.springframework.web.servlet.support.AbstractAnnotationConfigDispat
cherServletInitializer;

public class WebAppInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() { //Shared by whole
application
        return new Class[] { AppConfig.class };
    }

    @Override
    protected Class<?>[] getServletConfigClasses() { //specific to
this servlet: Handler Mapping, view resolver config, controllers etc
        return null;
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```

4️⃣ Create a Controller

```
package com.example.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;
```

```
@Controller
public class HelloController {

    @GetMapping("/")
    @ResponseBody
    public String hello() {
        return "Hello from Spring MVC with Embedded Tomcat!";
    }
}
```

5️⃣ Create the Embedded Tomcat Launcher

Here's the key part — start Tomcat from your `main()` method:

```
package com.example;

import org.apache.catalina.Context;
import org.apache.catalina.startup.Tomcat;

import java.io.File;

public class EmbeddedTomcatApp {

    public static void main(String[] args) throws Exception {
        Tomcat tomcat = new Tomcat();
        tomcat.setPort(8080);

        // Base directory for Tomcat temp files
        tomcat.setBaseDir("temp");

        // Create context - point to current directory for simplicity
        Context context = tomcat.addContext("", new
File(".").getAbsolutePath());

        // Add the dispatcher servlet programmatically
        Tomcat.addServlet(context, "dispatcher",
            new org.springframework.web.servlet.DispatcherServlet(
```

```
                new
org.springframework.web.context.support.AnnotationConfigWebApplication
Context() {{
                    register(com.example.config.AppConfig.class);
                }}
            )
        );
        context.addServletMappingDecoded("/", "dispatcher");

        tomcat.start();
        tomcat.getServer().await();
    }
}
```

6️⃣ Package and Run

Maven:

```
mvn clean package
```

Run:

```
java -jar target/your-app.jar
```

Test:

```
http://localhost:8080/
```

Points to note:

- You manually configure the `DispatcherServlet`.
- You explicitly create and configure the embedded Tomcat in `main()`.
- You manage everything (servlet mappings, base dirs, context).

Spring Boot

- “Here’s a fully-built kitchen, trained staff, ready-to-use menu, and all utilities connected. You just bring your recipes (your app’s business logic).”

Without Spring Boot (just using normal Java and Spring Framework):

- You have to configure **a lot**:
    - Set up web server (like Tomcat)
    - Connect database manually
    - Decide on project folder structure
    - Write lots of XML config files
- It’s like building a restaurant **brick by brick**.

With Spring Boot:

- Most of this is **already done**.

- You can start serving food (features) **immediately**.

---

3 — How it works

- **Spring Boot Starter Packs** → Pre-mixed masala packets.
  (Example: `spring-boot-starter-web` is a masala packet for making a web app — it has all the spices and ingredients you need: Spring MVC, Tomcat, JSON handling, etc.)

- **Embedded Server** → Your kitchen is already running inside the shop (no need to install a separate oven/stove).

- **Auto-Configuration** → Like a smart kitchen that adjusts itself when you bring in new recipes.
  (Example: If you bring rice, it'll automatically bring out the rice cooker.)

Earlier:

1) We had to manage lot of dependencies in Maven pom.xml

   Spring-web 5.4.1 may not be compatible with hibernate (orm) : 3.5.6

   Spring boot is a framework: It provides u starter dependencies

   Parent pom: 5.4.1

   Spring–jdbc

   Spring-orm

   Spring-core

   Spring-web

   Spring Tool Suite

   1) Provides us starter pom, based on which it will ensure dependencies for diff spring modules to be compatible with each other.

2) Embedding tomcat into our application, so we do not need a separate tomcat server. And using spring-boot-maven-plugin, I can create an executable jar file instead of having to deploy my application as a war file.
3) Any web application needs web.xml, In web.xml we configure the class DispatcherServlet mapped to what url should be handled by spring container.

   In springboot, it automatically creates the DispatcherServlet mapping for u. You don't have to do it

4) Provides you minimal configuration option. In application.properties, u can fine tune the parameters.
5) When i start my application, spring will try to autoconfigure some beans/ objects

   Datasource, DispatcherServlet, Mapping for DispatcherServlet, InternalResourceViewResolver…

   You need to provide only necessary details in property file to help it create those objects

   1) @SpringBootApplication:

      @SpringBootConfiguration : @Configuration: Bean classes

      @ComponentScan: What packages to look for stereotype annotations for creating the bean class objects

      => Controller: @Controller

      => Service: @Service

      => Repository: @Repository

      => util : @Component

      @ComponentScan(basePackages="com.example")

      Will try to create bean for classes inside this folder and subfolders

      @EnableAutoConfiguration: Spring is creating some beans for you automatically when you start your application:

      => Datasource

=> DispatcherServlet

=> ViewResolver

=> HandlerMapping

=> SessionFactory…

@Bean: Using this annotation in ur configuration class-> objects of the class

1) Add jpa library to your project
2) Create the interface for repository, extend JpaRepository interface

Optional:

1️⃣ What is Spring Data JPA?

It reduces boilerplate by letting you define **repository interfaces** instead of writing SQL or entity manager code.

Under the hood, it uses **Hibernate** (by default) as the JPA provider.

2️⃣ Basic Flow

Here's what you'll typically have:

```
Entity  <-- maps to DB table

Repository <-- handles CRUD automatically

Service  <-- business logic

Controller <-- REST or MVC endpoint
```

Spring Data JPA auto-generates SQL for CRUD methods based on method names or JPQL.

3️⃣ Step-by-Step Setup

Step 1: Add Maven/Gradle Dependencies

If you're using Maven:

```
<dependencies>
    <!-- Spring Boot Starter Data JPA -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <!-- Database driver (example: MySQL) -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

Spring Boot Starter Data JPA brings Hibernate + JPA dependencies automatically.

Step 2: Configure Database in `application.properties`

```
spring.datasource.url=jdbc:mysql://localhost:3306/testdb
spring.datasource.username=root
```

```
spring.datasource.password=yourpassword

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver


spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Di
alect
```

- `ddl-auto=update` lets Hibernate auto-create/update tables.
- `show-sql=true` logs SQL queries to console.

Step 3: Create an Entity Class

```
import jakarta.persistence.*;


@Entity

@Table(name = "users")

public class User {


    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;


    @Column(nullable = false)

    private String name;
```

```
    private String email;

    // Getters & Setters
}
```

Here:

- `@Entity` → Marks class as JPA entity (maps to table).
- `@Table` → Optional, sets custom table name.
- `@Id` → Primary key.
- `@GeneratedValue` → Auto-generates ID.
- `@Column` → Defines column properties.

Step 4: Create a Repository Interface

```
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {
    // You can define custom query methods here
    User findByEmail(String email);
}
```

- `JpaRepository<User, Long>` gives you CRUD methods like:
    - `save()`
    - `findById()`
    - `findAll()`
    - `deleteById()`
- Spring will generate implementation at runtime.

Step 5: Service Layer (Optional but Recommended)

```
import org.springframework.stereotype.Service;

import java.util.List;


@Service

public class UserService {


    private final UserRepository userRepository;


    public UserService(UserRepository userRepository) {

        this.userRepository = userRepository;

    }


    public List<User> getAllUsers() {
```

```
        return userRepository.findAll();
    }

    public User saveUser(User user) {
        return userRepository.save(user);
    }
}
```

- Good for keeping business logic separate from controllers.

Step 6: Controller Layer

```
import org.springframework.web.bind.annotation.*;
import java.util.List;

@RestController
@RequestMapping("/users")
public class UserController {

    private final UserService userService;

    public UserController(UserService userService) {
        this.userService = userService;
```

```
    }

    @GetMapping
    public List<User> getUsers() {
        return userService.getAllUsers();
    }

    @PostMapping
    public User createUser(@RequestBody User user) {
        return userService.saveUser(user);
    }
}
```

Now you have:

- `GET /users` → fetch all users
- `POST /users` → create a new user

Step 7: Run the App

If you have `@SpringBootApplication` in your main class, just run it and test:

```
GET  http://localhost:8080/users

POST http://localhost:8080/users

Content-Type: application/json
```

```
{
    "name": "Alice",
    "email": "alice@example.com"
}
```

✅ **Advantages of Spring Data JPA**

- No need to manually implement DAO methods.
- Strong naming conventions for queries (`findByName`, `findByEmailAndStatus`).
- Easy integration with pagination, sorting, and custom JPQL queries.

```
import org.springframework.data.jpa.repository.JpaRepository;
import java.util.List;

public interface UserRepository extends JpaRepository<User, Long> {

    // Find user by exact name
    List<User> findByName(String name);

    // Find user by email
    User findByEmail(String email);

    // Find all users whose name starts with given letters
    List<User> findByNameStartingWith(String prefix);
```

```
    // Find all users whose email contains something
    List<User> findByEmailContaining(String keyword);

    // Find by name and email
    User findByNameAndEmail(String name, String email);
}
```

5️⃣ How Method Names Work

Spring Data JPA reads your method name like English:

findBy → means “SELECT … WHERE …”

findByName → WHERE name = ?

findByNameAndEmail → WHERE name = ? AND email = ?

findByEmailContaining → WHERE email LIKE %?%

findByNameStartingWith → WHERE name LIKE ?%

It parses the method name and automatically builds the query.

6️⃣ Using the Repository

```
@Service
```

```
public class UserService {

    @Autowired
    private UserRepository repo;

    public void demo() {
        // Save a user
        repo.save(new User(null, "Alice", "alice@example.com"));

        // Find by name
        List<User> users = repo.findByName("Alice");

        // Find with email keyword
        List<User> gmailUsers = repo.findByEmailContaining("gmail");
    }
}
```

7️⃣ When You Need Complex Queries

If method naming is too long or not possible, you can:

- Use `@Query` annotation for JPQL:

```
@Query("SELECT u FROM User u WHERE u.name = :name AND u.email LIKE
%:domain%")

List<User> search(@Param("name") String name, @Param("domain") String
domain);
```

If your repository interface extends `JpaRepository<Entity, ID>`, you automatically get these methods:

---

1. Create / Insert / Update

```
<S extends T> S save(S entity)
```

- **If the ID is `null`** → inserts a new row.
- **If the ID exists** → updates the existing row.

Example:

```
userRepository.save(new User("John", "john@mail.com"));
```

---

2. Read

- **Get all rows**:

```
List<T> findAll();
```

- **Get by ID**:

```
Optional<T> findById(ID id);
```

- **Count total rows**:

```
long count();
```

---

3. Delete

- **Delete by ID**:

```
void deleteById(ID id);
```

- **Delete by entity**:

```
void delete(T entity);
```

- **Delete all**:

```
void deleteAll();
```

---

4. Exists check

```
boolean existsById(ID id);
```

---

5. Extra (Sorting & Paging)

If you extend `JpaRepository`, you can use:

```
List<T> findAll(Sort sort);

Page<T> findAll(Pageable pageable);
```

✅ **Key takeaway in layman terms**:
You just extend:

```
public interface UserRepository extends JpaRepository<User, Long> { }
```

…and you instantly get methods like:

- `save()` → insert/update
- `findAll()` → read all
- `findById()` → read one
- `deleteById()` → delete one
- `count()` → count rows

No SQL needed — JPA does it for you.

https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html