1️⃣ `@Controller` — what it *really* is

Definition

```
@Controller
public class MyController { }
```

- Marks the class as a **Spring MVC web controller**
- Designed for **server-side rendered views** (HTML, JSP, Thymeleaf, etc.)
- Returns **logical view names**, not raw data

Under the hood

`@Controller` is a specialization of:
`@Component`

So:

- It becomes a Spring bean
- It is detected by **component scanning**
- It participates in Spring MVC's **DispatcherServlet flow**

2️⃣ `@RestController` — what makes it special

```
@RestController
public class MyRestController { }
```

This is literally:

```
@Controller
@ResponseBody
```

Meaning:

- Every method automatically serializes return values
- Output goes directly to the **HTTP response body**
- No view resolution

💡 **This is the biggest difference.**

3️⃣ Request Flow (VERY important)

🔹 With `@Controller`

```
@GetMapping("/home")
public String home() {
    return "index";
}
```

Flow:

1. Request hits `DispatcherServlet`
2. Finds matching controller method
3. Method returns `"index"`
4. **ViewResolver** kicks in
5. `index.html` / `index.jsp` rendered
6. HTML sent to browser

📌 If no view resolver → ❌ error

🔹 With `@RestController`

```
@GetMapping("/users")
public List<User> getUsers() {
    return userService.findAll();
}
```

Flow:

1. Request hits `DispatcherServlet`
2. Finds method
3. Return value → **HttpMessageConverter**
4. Converted to JSON / XML
5. Written directly to response body

📌 ViewResolver is **skipped**

4️⃣ @ResponseBody — the real switch

In @Controller

```
@Controller
public class DemoController {

    @GetMapping("/hello")
    @ResponseBody
    public String hello() {
        return "Hello";
    }
}
```

✅ This behaves **exactly like a REST endpoint**

Without `@ResponseBody`:

- Spring thinks `"Hello"` is a **view name**
- Tries to find `Hello.html`

5️⃣ Content Negotiation (REST magic)

`@RestController` uses **content negotiation**:

Based on:

- `Accept` header
- Available `HttpMessageConverters`

Example:

```
Accept: application/json
```

Spring chooses:

- `MappingJackson2HttpMessageConverter`

You can return:

```
User
List<User>
Map<String, Object>
ResponseEntity<?>
```

6️⃣ `ResponseEntity` — full HTTP control

```
@GetMapping("/user/{id}")
public ResponseEntity<User> getUser(@PathVariable Long id) {
    return ResponseEntity
            .status(HttpStatus.OK)
            .header("X-App", "Demo")
            .body(user);
}
```

Control:

- HTTP status
- Headers
- Body

🔥 **This is REST done right.**

7️⃣ Exception Handling Differences

Works for both:

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(NotFoundException.class)
    public ResponseEntity<String> handleNotFound() {
        return ResponseEntity.status(404).body("Not Found");
    }
}
```

But:

- `@ControllerAdvice` → returns views
- `@RestControllerAdvice` → returns JSON

8️⃣ Validation behavior

```
@PostMapping("/users")
public User save(@Valid @RequestBody User user) {
    return user;
```

```
}
```

If validation fails:

- `MethodArgumentNotValidException`
- Auto 400 response
- JSON error (REST)
- Error page (MVC)

9️⃣ Common mistakes (exam & interview favorites 😈)

❌ Returning String in `@RestController`

```
return "hello";
```

➡️ Response body = `"hello"`

❗ Not a view

❌ Forgetting `@ResponseBody` in `@Controller`

```
return user;
```

➡️ Spring tries to find a view named `User`

❌ Mixing View + REST in same controller

Technically allowed, **architecturally bad**

🔟 When to use which?

Use Case	Annotation
HTML pages	`@Controller`
REST APIs	`@RestController`
Microservices	`@RestController`
Thymeleaf/JSP	`@Controller`
SPA backend (React/Angular)	`@RestController`

Interview One-Liner

`@RestController` is a convenience annotation that combines `@Controller` and `@ResponseBody` to eliminate view resolution and directly write data to the HTTP response body.

What is `DispatcherServlet`?

`DispatcherServlet` is the *Front Controller* of Spring MVC.
Every HTTP request in a Spring MVC / Spring Boot web app goes through `DispatcherServlet` first.
It:

- Receives **all incoming requests**
- Finds the **right controller**
- Invokes it
- Handles **responses, views, exceptions**

Think of it as **traffic police** 🚦 for your web application.

Why does Spring need it?

Without `DispatcherServlet`:

- Each controller would need its own servlet
- Routing logic would be duplicated
- No centralized exception handling
- No consistent lifecycle

Spring solves this with **Front Controller Pattern**.

Where does it live?

- It is a **Servlet**
- Extends:

`HttpServlet → FrameworkServlet → DispatcherServlet`

In **Spring Boot**:

- Automatically registered
- Mapped to `/` (all requests)
- No `web.xml` needed

🧬 Core Components used by `DispatcherServlet`

DispatcherServlet does almost nothing by itself — it **delegates**.

Component	Responsibility
`HandlerMapping`	Finds which controller method should handle the request
`HandlerAdapter`	Invokes the controller method
`HandlerInterceptor`	Pre/post processing (auth, logging)
`ViewResolver`	Resolves logical view names
`HttpMessageConverter`	Converts Java ↔ JSON/XML
`ExceptionResolver`	Handles exceptions

🚶 Full Request Flow (VERY IMPORTANT)

Let's trace a request:
`GET /users/1`

Step-by-step:

1️⃣ Request arrives

Browser → Tomcat → `DispatcherServlet`

2️⃣ HandlerMapping

`DispatcherServlet` asks:
"Who can handle `/users/1`?"
Examples:

- `RequestMappingHandlerMapping`
- `@GetMapping("/users/{id}")`

3️⃣ HandlerAdapter

Once found:

“How do I invoke this handler?”

- Calls controller method
- Handles method parameters (`@PathVariable`, `@RequestBody`, etc.)

4️⃣ Controller execution

Your code runs:

```
@GetMapping("/users/{id}")
public User getUser(@PathVariable Long id) {
    return userService.find(id);
}
```

5️⃣ Return value handling

If `@RestController`:

- Uses `HttpMessageConverter`
- Converts `User` → JSON
- Writes to response body

If `@Controller`:

- Interprets return as **view name**
- Goes to `ViewResolver`

6️⃣ View Resolution (MVC only)

```
return "home";
```

- ViewResolver → `home.html`
- Template engine renders HTML

7️⃣ Response sent back

Response → Tomcat → Browser

🧩 Diagram in words

```
Client
  ↓
DispatcherServlet
  ↓
```

```
HandlerMapping
  ↓
HandlerAdapter
  ↓
Controller
  ↓
(HttpMessageConverter / ViewResolver)
  ↓
Response
```

🚨 Interceptors (before & after controller)

`HandlerInterceptor`

Execution order:

1. `preHandle()`
2. Controller method
3. `postHandle()`
4. `afterCompletion()`

Used for:

- Authentication
- Logging
- Metrics
- Locale handling

⚠️ Exception Handling Flow

If controller throws exception:

1. `DispatcherServlet`
2. `HandlerExceptionResolver`
3. `@ExceptionHandler`
4. `@ControllerAdvice` / `@RestControllerAdvice`

🔍 How Spring Boot wires it automatically

Spring Boot:

- Creates `DispatcherServlet` bean

- Registers it with embedded Tomcat
- Auto-configures:
    - Handler mappings
    - Converters
    - Resolvers

You almost **never touch it directly**.

🧠 Interview Kill-Shot Answer

DispatcherServlet is the central front controller in Spring MVC that intercepts all HTTP requests, delegates them to appropriate controllers via handler mappings and adapters, and coordinates view resolution, message conversion, and exception handling.