Spring Boot High-Level Application Architecture

The Dev Project

Spring Boot Initialization

START

Start Application @SpringBootApplication with main()

1
1. Start Application (@SpringBootApplication with main())

The **@SpringBootApplication** annotation marks the main class of your Spring Boot application.
It combines three crucial annotations: **@Configuration**, **@EnableAutoConfiguration**, and **@ComponentScan**, serving as the entry point of the application.

Initialize Spring Application Context

2
2. Initialize Spring Application Context

The **Spring Application Context** is the core of the Spring Framework, responsible for managing the lifecycle and configuration of application beans. During initialization, Spring scans for components, reads configurations, resolves dependencies, and prepares beans for use.
In a Spring Boot application, the application context is initialized automatically when **SpringApplication.run()** is called.

Load Configurations (application.properties, etc.)

3
3. Load Configurations (application.properties, etc.)

Spring Boot simplifies configuration management by loading settings from external configuration files such as **application.properties** or **application.yml**. These files are located in the **src/main/resources** directory by default. The configuration values in these files define application-specific properties, such as server ports, database settings, and logging levels.

Perform Auto-Configuration: Configure Beans Based on Classpath

4
4. Perform Auto-Configuration: Configure Beans Based on Classpath

Spring Boot's auto-configuration feature automatically configures beans based on the libraries present in the classpath. This behavior is enabled by the **@EnableAutoConfiguration** annotation, which is part of the **@SpringBootApplication** annotation.

Component Scanning: Scan for Beans and Components

5
5. Perform Auto-Configuration: Configure Beans Based on Classpath

Component scanning in Spring Boot is a process where the framework automatically detects and registers beans, components, and configurations within the application context. It identifies classes annotated with **@Component**, **@Service**, **@Repository**, and **@Controller**.

Component Scanning: Scan for Beans and Components

5
5. Component Scanning: Scan for Beans and Components

Component scanning in Spring Boot is a process where the framework automatically detects and registers beans, components, and configurations within the application context. It identifies classes annotated with **@Component**, **@Service**, **@Repository**, and **@Controller**.

Create and Wire Beans: Dependency Injection

6
6. Create and Wire Beans: Dependency Injection

Spring uses **Dependency Injection** (DI) to manage the creation and lifecycle of beans and resolve dependencies between them. Beans are objects managed by the Spring container, created and injected at runtime. Spring Boot automates this process using annotations such as **@Component**, **@Service**, **@Repository**, and **@Controller** for bean creation and **@Autowired** or constructor-based injection for wiring dependencies.

Initialize DispatcherServlet: Setup Front Controller

7
7. Initialize DispatcherServlet: Setup Front Controller

The **DispatcherServlet** is the central front controller in Spring MVC. It is automatically configured in a Spring Boot application when **spring-boot-starter-web** is included in the dependencies. The **DispatcherServlet** handles incoming HTTP requests, dispatches them to the appropriate handlers (controllers), and prepares the HTTP response.

Setup Embedded Server (Tomcat/Jetty)

8
8. Setup Embedded Server (Tomcat/Jetty)

Spring Boot applications come with an embedded web server, such as Tomcat, Jetty, or Undertow, allowing them to run independently without requiring external server installations. By default, Spring Boot uses **Tomcat** as the embedded server, but you can switch to **Jetty** or **Undertow** by including the respective dependencies.

Start Embedded Server: Begin Listening for Requests

9
9. Start Embedded Server: Begin Listening for Requests

Once the embedded server (e.g., Tomcat, Jetty, or Undertow) is set up and initialized, it begins listening for incoming HTTP requests on the configured port (default: 8080). This step finalizes the server's readiness to handle client requests, routing them to the appropriate endpoints via the **DispatcherServlet**.

Ready to handle request

At this moment, our application is live and ready to handle any incoming request.

HTTP Request Handling Sequence

Listen on Configured Port (e.g., 8080)

1
1. Listen on Configured Port (e.g., 8080)

When the embedded server (e.g., Tomcat, Jetty, or Undertow) starts in a Spring Boot application, it listens for incoming HTTP requests on a configured port. By default, this port is 8080, but it can be customized through the application configuration (**application.properties** or **application.yml**) or via command-line arguments.

Receive HTTP Request: Client Sends Request

2
2. Receive HTTP Request: Client Sends Request

When a client (such as a web browser, mobile app, or API client) sends an HTTP request, the Spring Boot embedded server (e.g., Tomcat, Jetty, or Undertow) receives the request on the configured port. The server processes the request and forwards it to the **DispatcherServlet**, which acts as the front controller in Spring MVC.

DispatcherServlet Receives Request: Front Controller

3
3. DispatcherServlet Receives Request: Front Controller

The DispatcherServlet is the central component of the Spring MVC architecture, acting as the front controller. It is responsible for handling all incoming HTTP requests and delegating them to appropriate handlers (controllers) based on mappings.

HandlerMapping Determines Controller: Find Appropriate Handler

4
4. HandlerMapping Determines Controller: Find Appropriate Handler

In Spring MVC, the **HandlerMapping** is responsible for determining which controller or handler method should process an incoming HTTP request. When a request is received by the **DispatcherServlet**, it consults the **HandlerMapping** to identify the appropriate handler based on the request path, HTTP method, headers, or other criteria.

Invoke Controller Method: Execute Endpoint Logic

5
5. Invoke Controller Method: Execute Endpoint Logic

Once a request is routed by the HandlerMapping to the appropriate controller, the corresponding controller method is invoked to execute the business logic for the endpoint. Controller methods are annotated with mappings like **@GetMapping**, **@PostMapping**, and **@RequestMapping** to specify the HTTP method and request path they handle.

Execute Business Logic: Service Layer Processing

6
6. Execute Business Logic: Service Layer Processing

The service layer in a Spring Boot application is responsible for executing the core business logic. It acts as an intermediary between the controller and the data access layer (e.g., repositories). The service layer encapsulates complex logic, making the application modular, reusable, and testable.
Services are typically annotated with **@Service** to indicate that they hold business logic.

Prepare Response: Assemble Data to Send Back

7
7. Prepare Response: Assemble Data to Send Back

After executing the business logic in the service layer, the controller prepares a response to send back to the client. This step involves assembling the response data, determining the appropriate format (e.g., JSON, XML, or plain text), and setting HTTP status codes and headers as needed.
Spring Boot simplifies this process using:

- Annotations like **@ResponseBody** or **@RestController** to directly return data as JSON or XML.
- **ResponseEntity** for fine-grained control over the response, including headers and status codes.

Send Response to Client: HTTP Response Returned

8
8. Send Response to Client: HTTP Response Returned

After the controller prepares the response, Spring Boot's DispatcherServlet sends the HTTP response back to the client. This response includes the status code, headers, and body (if applicable). The response can be in various formats, such as JSON, XML, plain text, or HTML, depending on the client's request and server configurations.

This step finalizes the request-response lifecycle, delivering the result of the HTTP request back to the client application.

Client Receives Response: Display or Process Data

9
9. Send Response to Client: HTTP Response Returned

After the server sends the HTTP response, the client application processes the response data. The client may display the data to the user (e.g., in a web or mobile UI) or use it for further processing, such as updating internal states or triggering additional actions.
Clients typically process:

1. **HTTP Status Code**: Determines success or failure.
2. **Response Body**: Contains the requested data, often in JSON or XML format.
3. **Headers**: Provide metadata such as caching, content type, and custom server information.

Top 5 HTTP Status Codes Cheat Sheet

1. 200 OK
Meaning: The request was successful, and the server returned the requested data.

2. 201 Created
Meaning: The request resulted in the creation of a new resource.

3. 400 Bad Request
Meaning: The server could not process the request due to invalid client input.

4. 401 Unauthorized
Meaning: Authentication is required, but the client failed to provide valid credentials.

5. 404 Not Found
Meaning: The server could not find the requested resource.