`@Controller`

- This annotation marks a class as a **Spring MVC controller**.
- It is used to **handle web requests** and return **views (like HTML pages)** in traditional web applications (like JSP or Thymeleaf).
- Methods in this class typically return the **name of a view**, not raw data.

`@RestController`

- A specialized version of `@Controller`.
- Combines `@Controller` and `@ResponseBody`.
- Used in **REST APIs** where you want to return **JSON or XML** instead of views.
- All methods in this class will return **data directly to the client**.

`@SpringBootApplication`

- A **meta-annotation** that includes:
  - `@Configuration`: Marks the class as a Spring config class.
  - `@EnableAutoConfiguration`: Tells Spring Boot to configure the app automatically.
  - `@ComponentScan`: Automatically scans and registers beans in the package.
- It is placed on the **main class** to mark the entry point of a Spring Boot app.

`@RequestMapping`

- Used to **map web requests to specific handler methods** or controller classes.
- Can be used at the class level (base path) or method level (specific paths).
- Supports all HTTP methods (GET, POST, PUT, DELETE, etc.).

`@RequestParam`

- Binds a **URL query parameter** to a method argument.
- Useful for reading parameters like `/search?name=hari` or `/page?size=10`.

`@ResponseBody`

- Tells Spring to **return the method result directly as the HTTP response body**.
- Used in REST APIs to send data (like JSON) instead of view names.
- Automatically serializes Java objects to JSON (or XML).

`@RequestBody`

- Binds the **HTTP request body** to a method argument.
- Useful when sending **JSON or XML in a POST/PUT request**.
- Spring automatically **converts JSON into Java objects**.

`@GetMapping`

- Shortcut for `@RequestMapping(method = RequestMethod.GET)`
- Used for handling **GET requests**, which are used to **retrieve data**.

`@PostMapping`

- Shortcut for `@RequestMapping(method = RequestMethod.POST)`
- Used for handling **POST requests**, which are used to **submit or create data**.

What is a REST API?

REST API stands for **Representational State Transfer Application Programming Interface**.

It is a **way for different software systems to communicate with each other** over the internet using **standard HTTP methods** like `GET`, `POST`, `PUT`, `DELETE`, etc.

A **REST API** is like a **messenger** that allows your **frontend** (like a mobile app or website) to **talk to a backend server** to **get or send data**.

Concept	Meaning
Client	The app or browser that sends requests (e.g., a mobile app)
Server	The backend service that processes and sends responses
Resource	Any data object — e.g., user, product, file
Endpoint	A URL where the resource can be accessed
HTTP Methods	Actions that can be performed (GET, POST, PUT, DELETE)
JSON/XML	Common data formats used in REST communication

What is JSON?

JSON stands for **JavaScript Object Notation**.

It is a **lightweight, text-based format** for **storing and exchanging data** between systems — especially between a **client (like a web browser or mobile app)** and a **server**.

JSON is just **data in a text format** that looks like a **JavaScript object**. It is easy for both **humans to read** and **machines to parse**.

Where is JSON Used?

- In **REST APIs** to send/receive data
- In **AJAX** calls for dynamic web pages
- In **configuration files** (`package.json`, `appsettings.json`)
- In **database storage** (MongoDB uses JSON-like documents)

What is a Web Service?

A **web service** is a **software system** designed to **communicate with other systems** over a **network** (usually the internet) using **standard web protocols** like **HTTP**.

A **web service** is like an **online function** that one application can call over the internet to **send or receive data** from another application.

Type	Description
SOAP (Simple Object Access Protocol)	Uses XML, very strict structure, heavier and older
REST (Representational State Transfer)	Uses HTTP, JSON/XML, lightweight and commonly used

What is Spring Boot?

Spring Boot is an **open-source Java-based framework** built on top of the Spring Framework that is designed to **simplify the development of stand-alone, production-grade Spring applications**, including **web applications and RESTful web services**, by providing features like **auto-configuration**, **starter dependencies**, and an **embedded web server** with **minimal setup and configuration**.

Features of Spring Boot

- **Auto-Configuration**

Spring Boot provides **intelligent auto-configuration**, meaning it automatically configures your application based on the libraries you include. For example, if you add `spring-boot-starter-web`, Spring Boot will automatically set up the embedded Tomcat server, Spring MVC, and basic configurations for handling HTTP requests — without needing you to define anything manually.

- **Embedded Web Servers**

Spring Boot comes with **built-in web servers** like Tomcat, Jetty, or Undertow. This means you don't need to deploy your application as a WAR file to an external server. Instead, your application can run as a simple Java program (`java -jar app.jar`) and handle web requests directly. This simplifies deployment and testing.

- **Starter Dependencies**

To simplify dependency management, Spring Boot provides **starter packages**. These are pre-configured collections of commonly used libraries grouped together. For example, `spring-boot-starter-web` includes everything you need to build a web application, like Spring MVC, Jackson for JSON, and an embedded server. This avoids the hassle of manually adding multiple dependencies.

- **Production-Ready Features**

Spring Boot includes **Actuator**, which gives you built-in endpoints to monitor and manage your application in production. These endpoints can expose health status, metrics, application environment details, and more, helping with diagnostics and operations.

- **Spring Boot CLI**

Spring Boot includes a **Command-Line Interface (CLI)** that lets you run Spring applications using simple scripts without writing full Java classes. It's useful for quick prototyping or scripting.

- **Spring Initializr**
Spring Boot provides a web tool called **Spring Initializr** (https://start.spring.io), which helps you quickly generate a project with the required dependencies. You can choose your language, dependencies, and project type, and download a ready-to-use template in seconds.

- **No XML Configuration**
Unlike older Spring applications, Spring Boot avoids XML configuration. Instead, it relies on **annotations and Java-based configuration**, which makes the code cleaner and more readable. This reduces complexity and makes your application easier to maintain.

- **Fast Development**
Thanks to auto-configuration, starter dependencies, and embedded servers, Spring Boot allows you to **develop applications faster**. You can focus more on writing business logic instead of handling configuration, setup, or deployment issues.

- **Microservices Support**
Spring Boot is designed to support **microservice architecture**. It's lightweight and modular, which makes it perfect for building independent services that communicate with each other. When combined with **Spring Cloud**, it becomes a powerful toolkit for microservice development.

- **Custom Configuration**
Spring Boot allows you to easily customize application settings using `application.properties` or `application.yml` files. You can change server ports, database connections, logging levels, and more — all through simple key-value pairs, without changing your code.

- **Seamless Integration with Spring Ecosystem**
Spring Boot works perfectly with other Spring projects like **Spring Security**, **Spring Data JPA**, **Spring Cloud**, and more. You can plug them in easily and they'll be auto-configured based on your needs.

- **DevTools for Developer Experience**
Spring Boot provides **DevTools**, a set of developer-focused features like **automatic restarts**, **live reload**, and **debugging support**. These features help you test changes instantly during development without restarting the whole application manually.

Create a REST API Using Spring Boot

Step 1: Create a Spring Boot Project

You can use **Spring Initializr**:

- Go to: https://start.spring.io

- Select:
  - Project: Maven
  - Language: Java
  - Spring Boot Version: (latest stable)
  - Project Metadata:
    - Group: `com.example`
    - Artifact: `demo`
- Add Dependencies:
  - `Spring Web`
- Click **"Generate"** to download the project ZIP
- Extract it and open in your IDE (IntelliJ, Eclipse, or VS Code)

Step 2:Understand the Project Structure

```
src/
 └── main/
     ├── java/
     │   └── com/example/demo/
     │       └── DemoApplication.java
     └── resources/
         └── application.properties
```

Step 3: Create a REST Controller Class

Create a new class inside `com.example.demo` named `HelloController`.

Use the annotation `@RestController` and define a simple method using `@GetMapping`.

Example: (Conceptual - No code here)

- Mark the class as a REST controller.
- Define a method that handles a GET request (like `/hello`).
- The method returns a plain string or a JSON response.

Step 4: Run the Application

- Run the `DemoApplication` class.
- Spring Boot starts an embedded Tomcat server.
- You'll see something like:
  ```
  Tomcat started on port(s): 8080
  ```

Step 5: Test the API

- Open your browser or use **Postman**.

- Access:
  `http://localhost:8080/hello`
- You should see the response returned from your method.

Step 6: Add More API Endpoints

You can add more methods using:

- `@PostMapping` – for creating data
- `@PutMapping` – for updating data
- `@DeleteMapping` – for deleting data
- Use `@RequestParam` for query parameters and `@RequestBody` for JSON input

Step 7: Customize with application.properties (Optional)

```
server.port=9090
```