1\*\*Spring Boot\*\* is an extension of the Spring framework that simplifies the process of building production-ready Spring applications. It provides a set of tools and frameworks that reduce the need for extensive configuration, making it easier to get started with Spring-based projects.

### Key Features and Uses of Spring Boot:

1. \*\*Auto-Configuration:\*\*

- Spring Boot automatically configures your Spring application based on the dependencies you have in your project. This reduces the need for manually configuring beans, data sources, and other components.

2. \*\*Embedded Servers:\*\*

- Spring Boot comes with embedded servers like Tomcat, Jetty, or Undertow, which means you don’t need to deploy your application to an external server. You can run your Spring Boot application as a standalone Java application.

3. \*\*Opinionated Defaults:\*\*

- Spring Boot provides sensible defaults to reduce boilerplate code and configuration, allowing developers to focus on the business logic rather than configuration details.

4. \*\*Starter POMs:\*\*

- Spring Boot uses "starter" dependencies to group common dependencies together. For example, adding `spring-boot-starter-web` to your project automatically pulls in all the dependencies required to create a web application.

5. \*\*Production-Ready Features:\*\*

- Spring Boot includes various features to help in monitoring and managing applications in production. The Spring Boot Actuator, for instance, provides endpoints to check application health, metrics, and other monitoring information.

6. \*\*Microservices:\*\*

- Spring Boot is widely used to build microservices due to its simplicity and flexibility. It provides tools and libraries that support the creation of independent, deployable services that can communicate with each other.

7. \*\*Reduced Boilerplate:\*\*

- Spring Boot reduces the amount of boilerplate code needed, allowing developers to build applications quickly and with fewer lines of code.

### Common Use Cases:

- \*\*RESTful APIs\*\*: Spring Boot makes it easy to build RESTful APIs with minimal configuration.

- \*\*Microservices\*\*: Ideal for building microservices due to its lightweight nature and support for cloud-native development.

- \*\*Web Applications\*\*: You can build full-stack web applications using Spring Boot with built-in support for Spring MVC.

- \*\*Batch Processing\*\*: Supports the creation of batch processing applications.

- \*\*Data-Driven Applications\*\*: Integrates seamlessly with databases and provides tools for data access and manipulation.

Overall, Spring Boot is designed to simplify the development process, making it easier to create and deploy Spring applications quickly and efficiently.

**RE**presentational **S**tate **T**ransfer (REST) is an architectural style that defines a set of constraints to be used for creating web services. **REST API**is a way of accessing web services in a simple and flexible way without having any processing.

In Spring Boot, @RestController is a key annotation that simplifies the development of RESTful web services. It combines the functionality of @Controller and @ResponseBody, making it easier to build REST APIs.

**Definition of @RestController in Spring Boot:**

@RestController: An annotation in Spring Boot used to create RESTful web services. It marks a class as a controller where every method automatically serializes return objects into HTTP responses. Instead of returning a view, it returns the response body directly, usually as JSON or XML.

**Purpose of @RestController:**

**REST API Development**: @RestController is specifically designed to simplify the creation of RESTful web services by removing the need to annotate each method with @ResponseBody. It handles HTTP requests and returns data directly to the client, which is ideal for building APIs that communicate with front-end applications or other services.

Key Features:

**Automatic Conversion:** Converts the return values of methods to JSON or XML using Spring's HttpMessageConverter mechanism.

**No View Resolution**: Unlike traditional controllers that return a view (like a JSP page), @RestController methods return data directly, without going through a view resolver.

**Simplified Code:** By eliminating the need for @ResponseBody on each method, it reduces boilerplate code.

In Spring Boot, @RequestMapping is a versatile annotation used to map web requests to specific handler methods or classes. It can be applied at the class level or method level, allowing you to define the URL path that a method or controller will handle.

**Definition of @RequestMapping in Spring Boot:**

* **@RequestMapping**: An annotation in Spring Boot used to map HTTP requests to handler methods of MVC and REST controllers. It provides a way to specify the request path, HTTP method, request parameters, headers, and other attributes.

**Purpose of @RequestMapping:**

The main purpose of @RequestMapping is to create a mapping between a URL and a controller method. This allows the method to handle requests that match the specified URL pattern. It is used for routing incoming web requests to appropriate handler methods.

**Key Features:**

1. **Versatile Mapping**: Supports various attributes like URL paths, HTTP methods (GET, POST, etc.), headers, parameters, and more.
2. **Class-Level and Method-Level**: Can be used at the class level to define a base path and at the method level to specify additional paths or actions.
3. **Supports Multiple HTTP Methods**: Allows specifying one or more HTTP methods (e.g., GET, POST) that a handler method can process.

**Attributes of @RequestMapping:**

* **value or path**: Specifies the URL pattern to map.
* **method**: Specifies the HTTP method(s) (e.g., GET, POST, PUT, DELETE).
* **params**: Specifies request parameters that must be present for the method to be invoked.
* **headers**: Specifies HTTP headers that must be present for the method to be invoked.
* **produces and consumes**: Specifies the media types that the method produces or consumes.

**Summary:**

* **@RequestMapping** is a powerful and flexible annotation in Spring Boot used to define request-to-handler mappings.
* It can be customized with various attributes to precisely control how web requests are handled by your application.
* For specific HTTP methods, it’s often easier to use annotations like @GetMapping, @PostMapping, @PutMapping, and @DeleteMapping.

package com.in28minutes.springboot.learn\_spring\_boot;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class LearnSpringBootApplication {

public static void main(String[] args) {

SpringApplication.run(LearnSpringBootApplication.class, args);

}

}

**package** com.in28minutes.springboot.learn\_spring\_boot;

**import** java.util.Arrays;

**import** java.util.List;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.bind.annotation.RestController;

@RestController

**public** **class** CourseController

{

@RequestMapping("/courses")

**public** List<Course> retrieveAllCourses()

{

**return** Arrays.*asList*(

**new** Course(1 , "AWS" , "Author1"),

**new** Course(2 , "JAVA" , "Author2")

);

}

}

**package** com.in28minutes.springboot.learn\_spring\_boot;

**public** **class** Course

{

**private** **long** id;

**private** String name;

**private** String author;

**public** Course(**long** id, String name, String author) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.author = author;

}

**public** **long** getId() {

**return** id;

}

**public** String getName() {

**return** name;

}

**public** String getAuthor() {

**return** author;

}

@Override

**public** String toString() {

**return** "Course [id=" + id + ", name=" + name + ", author=" + author + "]";

}

}

**THE CODE WITH SPRING FRAME WORK NEEDS TO WRITE**

To convert the Spring Boot code into standard Spring (without Spring Boot) to create the same REST API, you will need to:

1. **Add Spring MVC configuration.**
2. **Manually configure the Spring context.**

Here's how you can do it in a standard Spring application:

1. **Create a Spring configuration file (applicationContext.xml)** to define the beans and configure the Spring MVC dispatcher servlet.
2. **Set up a web.xml** to configure the dispatcher servlet.

**SO SPRINGBOOT IS BETTER THAN SPRING FRAMEWORK.**

What's the Most Important Goal of Spring Boot?

\* Help you build PRODUCTION-READY apps QUICKLY

■ Build QUICKLY

\* Spring Initializr

\* Spring Boot Starter Projects

\* Spring Boot Auto Configuration

\* Spring Boot DevTools

■ Be PRODUCTION-READY

\* Logging

\* Different Configuration for Different Environments

\* Profiles, Configuration Properties

\* Monitoring (Spring Boot Actuator)

Exploring Spring Boot Starter Projects

\* I need a lot of frameworks to build application features:

■ Build a REST API: I need Spring, Spring MVC, Tomcat, JSON conversion...

■ Write Unit Tests: I need Spring Test, JUnit, Mockito, ...

\* How can I group them and make it easy to build applications?

■ Starters: Convenient dependency descriptors for diff. features

Spring Boot provides variety of starter projects:

■ Web Application & REST API - Spring Boot Starter Web (spring-webmvc, spring-web, spring-boot-starter-tomcat, spring-boot-starter-json)

■ Unit Tests - Spring Boot Starter Test

■ Talk to database using JPA - Spring Boot Starter Data JPA

■ Talk to database using JDBC - Spring Boot Starter JDBC

■ Secure your web application or REST API - Spring Boot Starter Security

\* I need lot of configuration to build Spring app:

■ Component Scan, DispatcherServlet, Data Sources, JSON Conversion, .

\* How can I simplify this?

■ Auto Configuration: Automated configuration for your app

\* Decided based on:

\* Which frameworks are in the Class Path?

\* What is the existing configuration (Annotations etc)?

\* Example: Spring Boot Starter Web

■ Dispatcher Servlet (DispatcherServletAutoConfiguration)

■ Embedded Servlet Container Tomcat is the default (EmbeddedWebServerFactoryCustomizerAutoConfiguration)

■ Default Error Pages (ErrorMvcAutoConfiguration)

I ■ Bean<->JSON

(JacksonHttpMessageConvertersConfiguration)

**DIFFERENT LOGIN LEVELS**

Managing App. Configuration using Profiles

\* Applications have different environments: Dev, QA, Stage, Prod, ...

\* Different environments need different configuration:

■ Different Databases

■ Different Web Services

\* How can you provide different configuration for different environments?

■ Profiles: Environment specific configuration

1.trace

2.debug

3.info

4.warning

5.error

6.off

**GET PRODUCTION READY WITH SPRING BOOT-2-CONFIGURATION PROPERTIES**

**package** com.in28minutes.springboot.learn\_spring\_boot;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.bind.annotation.RestController;

@RestController

**public** **class** CurrencyConfigurationController

{

@Autowired

**private** CurrencyServiceConfiguration configuration;

@RequestMapping("/currency-configuration")

**public** CurrencyServiceConfiguration retrieveAllCourses()

{

**return** configuration;

}

}

package com.in28minutes.springboot.learn\_spring\_boot;

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.stereotype.Component;

@ConfigurationProperties(prefix = "currency-service")

@Component

public class CurrencyServiceConfiguration

{

private String url;

private String username;

private String key;

public String getUrl() {

return url;

}

public void setUrl(String url) {

this.url = url;

}

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

public String getKey() {

return key;

}

public void setKey(String key) {

this.key = key;

}

}

spring.application.name=learn-spring-boot

logging.level.org.springframework=debug

spring.profiles.active=dev

currency-service.url=https://chatgpt.com/

currency-service.username=Sivaram

[currency-service.key=Siva@123](mailto:currency-service.key=Siva@123)

Simplify Deployment with Spring Boot Embedded Servers

\* How do you deploy your application?

■ Step 1: Install Java

■ Step 2: Install Web/Application Server

\* Tomcat/WebSphere/WebLogic etc

■ Step 3: Deploy the application WAR (Web Archive)

\* This is the OLD WAR Approach

\* Complex to setup!

**Embedded Server - Simpler alternative**

■ Step 1: Install Java

■ Step 2: Run JAR file

■ Make JAR not WAR (Credit: Joshi Long!)

■ Embedded Server Examples:

spring-boot-starter-tomcat

spring-boot-starter-jetty

spring-boot-starter-undertow

Understanding Spring Boot vs Spring MVC vs Spring

-28

\* Spring Boot vs Spring MVC vs Spring: What's in it?

■ Spring Framework: Dependency Injection

\* @Component, @Autowired, Component Scan etc..

\* Just Dependency Injection is NOT sufficient (You need other frameworks to build apps)

\* Spring Modules and Spring Projects: Extend Spring Eco System • Provide good integration with other frameworks (Hibernate/JPA, JUnit & Mockito for Unit Testing)

■ Spring MVC (Spring Module): Simplify building web apps and REST API

\* Building web applications with Struts was very complex

\* @Controller, @RestController, @RequestMapping("/courses")

■ Spring Boot (Spring Project): Build PRODUCTION-READY apps QUICKLY

\* Starter Projects - Make it easy to build variety of applications

\* Auto configuration - Eliminate configuration to setup Spring, Spring MVC and other frameworks!

\* Enable non functional requirements (NFRs):

\* Actuator: Enables Advanced Monitoring of applications

\* Embedded Server: No need for separate application servers!

\* Logging and Error Handling

\* Profiles and Configuration Properties