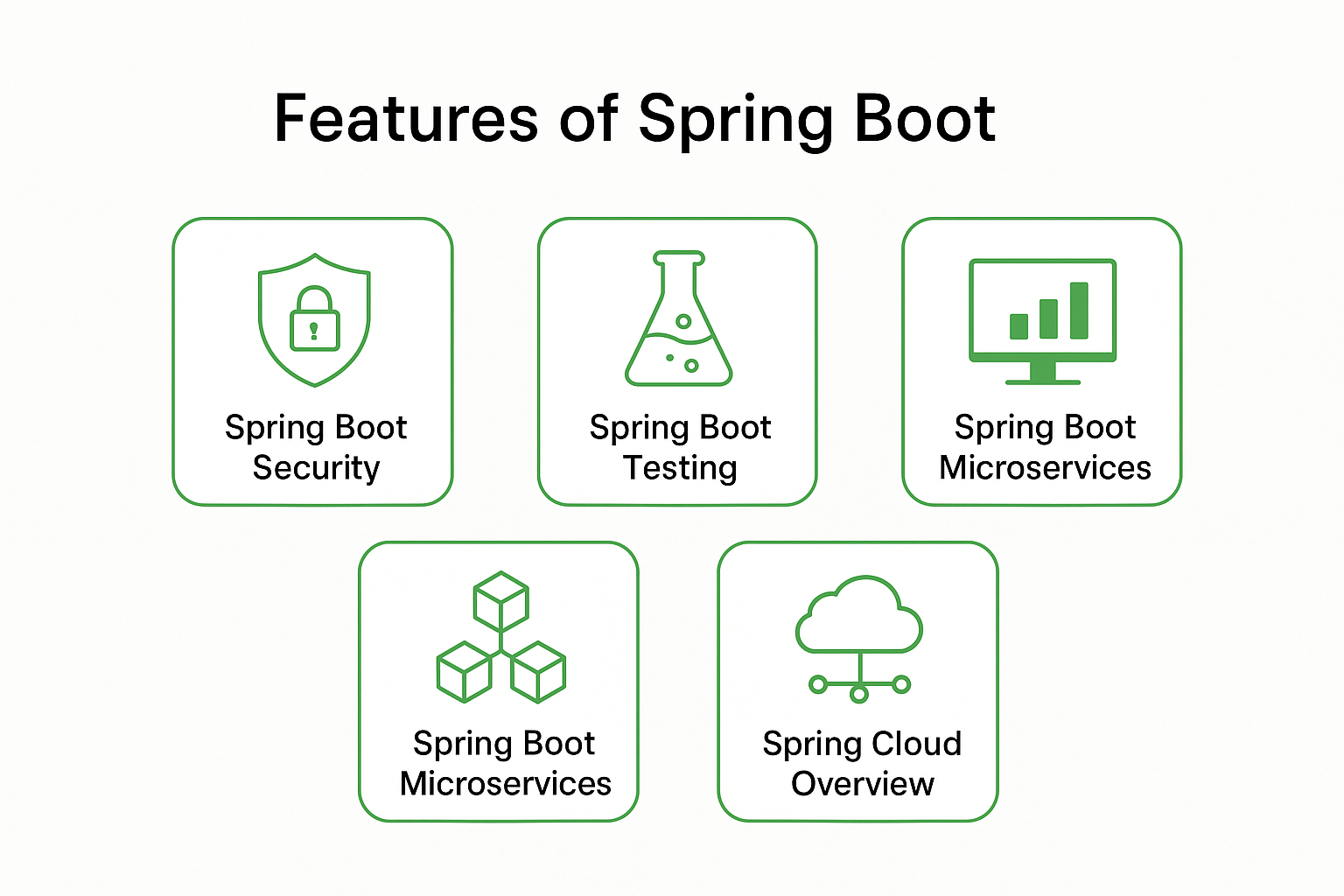
**UNIT-4**

**Features of Spring Boot: Spring Boot security, Spring Boot testing, Spring boot actuator, Spring boot microservices, Spring cloud overview.**

**Developing reactive APIs: Working with Spring WebFlux (Introducing Spring WebFlux, writing reactive controllers), Defining functional request handlers, testing reactive controllers (Testing GET requests, Testing POST requests, testing with a live server), Consuming REST APIs reactively (GETting resources, sending resources, deleting resources, handling errors, exchanging requests), Securing reactive web APIs (Configuring reactive web security, Configuring a reactive, user details service).**

**Textbook 1: Chapter 11:11.1 to 11.5**

****

**1. Introduction to Spring Boot**

**Spring Boot is a module of the Spring framework that simplifies application development by providing defaults for code and annotation configuration to quickly set up and run Spring applications.**

**Key Features:**

* **Auto-configuration**
* **Embedded servers (like Tomcat, Jetty)**
* **Production-ready features (health checks, metrics)**
* **Opinionated defaults to reduce boilerplate code**

**2. 🌐 Spring Boot Security**

**Overview:**

**Spring Boot Security is built on Spring Security. It provides authentication, authorization, and protection against common attacks.**

**Features:**

* **Auto-configuration of Spring Security components**
* **Form-based login by default**
* **Password encoding (BCrypt by default)**
* **Role-based access control**
* **JWT support for token-based authentication**
* **Method-level security using annotations like @PreAuthorize, @Secured**

**3. 🧪 Spring Boot Testing**

**Overview:**

**Spring Boot provides strong support for testing through integration with JUnit, Mockito, and Spring TestContext Framework.**

**Features:**

* **@SpringBootTest: Loads the full application context**
* **@WebMvcTest: Tests only Spring MVC components**
* **@DataJpaTest: Configures in-memory database and JPA components**
* **MockMvc: Allows testing of HTTP endpoints without running a full server**
* **Embedded databases for isolated test environments (H2, HSQLDB)**

**4. 📊 Spring Boot Actuator**

**Overview:**

**Spring Boot Actuator provides production-ready features such as metrics, health checks, and monitoring endpoints.**

**Key Endpoints:**

* **/actuator/health – Health status**
* **/actuator/metrics – Application metrics**
* **/actuator/env – Environment properties**
* **/actuator/loggers – Logging configuration**

**5. 🧩 Spring Boot Microservices**

**Overview:**

**Spring Boot is widely used to build microservices due to its lightweight and modular design.**

**Microservice Characteristics:**

* **Decentralized architecture**
* **Independent deployment**
* **RESTful communication**
* **Lightweight containers (e.g., Docker)**

**Tools:**

* **Spring Boot for creating services**
* **Spring Cloud for service discovery, load balancing**
* **Eureka, Ribbon, Zuul/Gateway, Config Server**

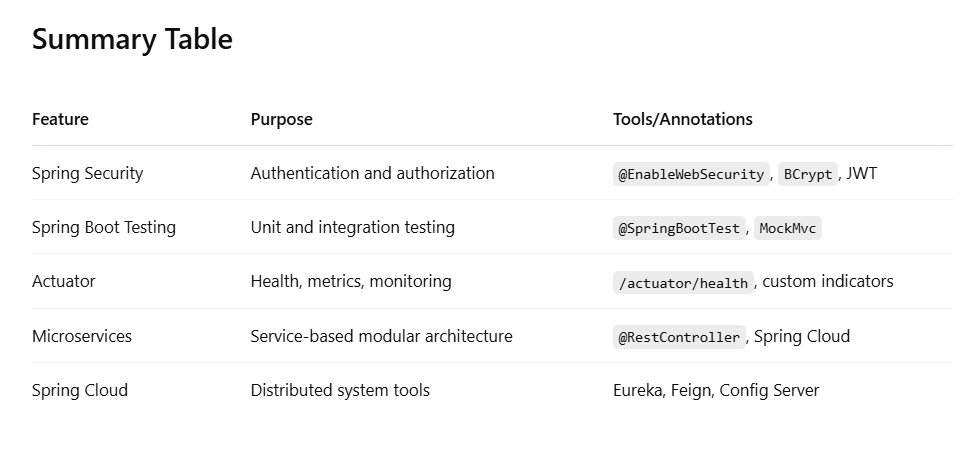
**6. ☁️ Spring Cloud Overview**

**Purpose:**

**Spring Cloud builds on Spring Boot to provide tools for developers to quickly build distributed systems.**

**Key Components:**

* **Eureka – Service Discovery**
* **Spring Cloud Config – Centralized configuration**
* **Feign – Declarative REST clients**
* **Ribbon – Client-side load balancing**
* **Hystrix – Fault tolerance (now replaced by Resilience4j)**
* **Gateway – API Gateway for routing**

****

**Developing Reactive APIs with Spring WebFlux**

**1. 🚀 Introducing Spring WebFlux**

**What is WebFlux?**

**Spring WebFlux is a reactive, non-blocking web framework introduced in Spring 5. It supports both annotation-based and functional programming models using Reactor (Mono & Flux).**

**Key Concepts:**

* **Reactive Streams: Asynchronous stream processing with non-blocking backpressure.**
* **Mono: 0..1 elements**
* **Flux: 0..N elements**
* **Built on Project Reactor and supports Netty, Undertow, or Tomcat**

**2. ✍️ Writing Reactive Controllers**

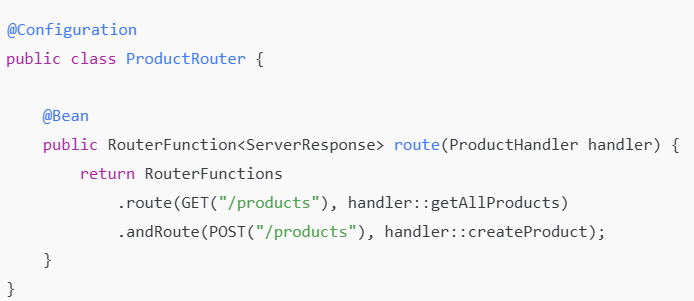
****

**Characteristics:**

* **Returns Mono<T> or Flux<T>**
* **Non-blocking data flow**
* **Integrates with Reactive MongoDB, R2DBC, etc.**

**3. 🛠️ Defining Functional Request Handlers**

**Spring WebFlux supports functional routing using RouterFunction and HandlerFunction.**

****

****

**4. ✅ Testing Reactive Controllers**

**a. Testing GET Requests**

****

**b. Testing POST Requests**

****

**c. Testing with Live Server**

Use @SpringBootTest(webEnvironment=WebEnvironment.RANDOM\_PORT) with WebTestClient.

**5. 🌐 Consuming REST APIs Reactively**

Use WebClient to perform non-blocking HTTP calls.

**a. GETting Resources**

WebClient.create()

.get()

.uri("/products")

.retrieve()

.bodyToFlux(Product.class);

**b. Sending Resources**

webClient.post()

.uri("/products")

.bodyValue(new Product("Phone", 500))

.retrieve()

.bodyToMono(Product.class);

**c. Deleting Resources**

webClient.delete()

.uri("/products/{id}", id)

.retrieve()

.toBodilessEntity();

**d. Handling Errors**

webClient.get()

.uri("/products")

.retrieve()

.onStatus(HttpStatus::isError, response -> Mono.error(new RuntimeException("API Error")));

**e. Exchanging Requests**

webClient.method(HttpMethod.PUT)

.uri("/products/{id}", id)

.bodyValue(product)

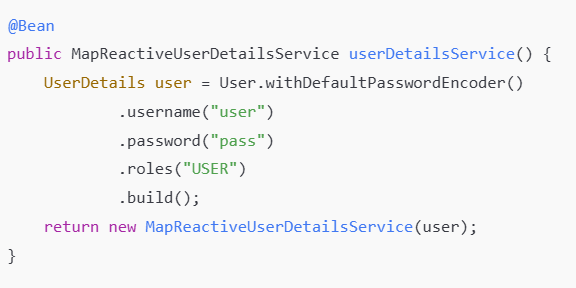
.exchangeToMono(response -> response.bodyToMono(String.class));

**6. 🔐 Securing Reactive Web APIs**

**a. Configuring Reactive Web Security**

****

**b. Reactive User Details Service**

****

****

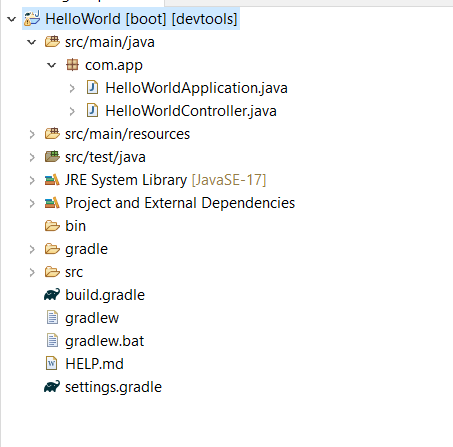
**Spring WebFlux Hello World Example**

**The**[**Spring Web Flux**](https://www.geeksforgeeks.org/basic-introduction-to-spring-webflux/)**provides different applications to developers to develop asynchronous, non-blocking web applications by using Mono, Flux, and other classes. In this article, we will be learning to write a basic Hello World program in Spring WenFlux.**

**Project Creation:**

* **Open STS (Spring Tool Suite) and then select New Project.**
* **After that, we will be redirected to a project creation screen, in Select Project Category like maven or gradle, choose any (Here we will be using Gradle).**
* **Then provide the project name, package name, and other details.**
* **Then select dependencies.**
* **Now click on finish.**

**Project Folder Structure:**

****

**Project Dependencies:**

**In this project, we have used below dependencies and the project category is gradle.**

dependencies {  
 implementation 'org.springframework.boot:spring-boot-starter-webflux'  
 developmentOnly 'org.springframework.boot:spring-boot-devtools'  
 testImplementation 'org.springframework.boot:spring-boot-starter-test'  
 testImplementation 'io.projectreactor:reactor-test'  
}

**Hello World Program in Spring WebFlux**

Here, we have provided a basic example for printing Hello World by using @RestController.

* For this, in main package, we have created one Java class named as HelloWorldController.
* After that, we have created one @GetMapping end point with name hello.
* The @GetMapping is used for creating API End points.
* While open with browser, this return output as Hello, World!.

**HelloWorldController class:**

**package com.app;**

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

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

**import reactor.core.publisher.Mono;**

**@RestController**

**public class HelloWorldController {**

**@GetMapping("/hello")**

**public Mono<String> hello() {**

**return Mono.just("Hello, World!"); *//prints Hello, World!***

**}**

**}**

**Main Class:**

**This is a main class of the project.**

**package com.app;**

**import org.springframework.boot.SpringApplication;**

**import org.springframework.boot.autoconfigure.SpringBootApplication;**

**@SpringBootApplication**

**public class HelloWorldApplication {**

***// Main Method***

**public static void main(String[] args) {**

**SpringApplication.run(HelloWorldApplication.class, args);**

**}**

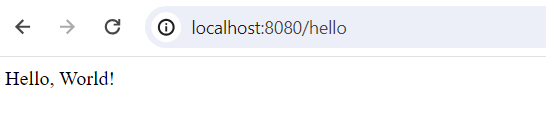
**}**

**After successfully running this project as Spring Boot App, open any browser then type below API End Point URL:**

**http://localhost:8080/hello**

**Output:**

**Below we can see the output in browser.**

****

**Explanation of the above Program:**

* **First, we have created one class in main package of project.**
* **The class name is HelloWorldController.**
* **After this, we have created one Get Mapping by using @GetMapping with parameter /hello.**
* **Next, we run this project as Spring Boot Application.**
* **After running, open browser then type the Get mapping URL that is provided in the above.**
* **Once hit that API end point, we got the expected output that is visible in the output screen.**

**Spring WebFlux** is a reactive web framework introduced in Spring Framework 5.0. It's designed to handle concurrency and backpressure efficiently, making it suitable for applications requiring high throughput and low latency. Spring Boot simplifies the development of WebFlux applications by providing auto-configuration and a streamlined setup process.

**WebFlux** operates on a non-blocking, event-driven architecture, contrasting with Spring MVC's traditional thread-per-request model. This allows WebFlux to handle more concurrent requests with fewer threads, improving resource utilization and scalability. It's particularly beneficial for I/O-bound operations, where waiting for external resources (like databases or network requests) can be a bottleneck.

**Key features of Spring WebFlux include:**

* **Reactive Programming:**

Built on Project Reactor, it uses Mono and Flux to represent asynchronous data streams.

* **Non-blocking I/O:**

Leverages non-blocking I/O operations to avoid tying up threads while waiting for responses.

* Functional and Annotation-based Styles:

Supports both functional routing and traditional annotation-based controller definitions.

* **WebClient:**

Provides a reactive HTTP client for making asynchronous requests to other services.

* **Integration with Spring Data:**

Enables reactive data access with databases like MongoDB and Cassandra.

Spring Boot simplifies WebFlux development by:

* **Auto-configuration:**

Automatically configures necessary components like the server (Netty by default), codecs, and reactive data repositories.

* **Starters:**

Offers convenient starter dependencies, such as spring-boot-starter-webflux, to include all required libraries.

* **Embedded Servers:**

Supports embedded servers like Netty, Tomcat, Jetty, and Undertow, with Netty being the default for reactive applications.

**When to consider using Spring WebFlux:**

* Applications requiring high concurrency and scalability.
* Services with many I/O-bound operations.
* Systems needing real-time updates or event-driven architectures.

**However, WebFlux may not be the best choice for:**

* Simple applications with minimal concurrency requirements.
* Projects where blocking I/O is acceptable and simpler to manage.
* Situations where many third-party libraries do not yet support reactive programming.