What is Spring Boot?

Spring Boot is an open-source Java framework used to create a Micro Service.It provides a faster way to set up and an easier, configure, and run both simple and web-based applications. Spring Boot, a module of the Spring framework and Embedded Servers. The main goal of Spring Boot is to reduce development, unit test, and integration test time and in Spring Boot, there is no requirement for XML configuration allowing developers to focus more on writing code for their applications..It facilitates **Rapid Application Development** (RAD) capabilities.

Spring Boot Actuator :

Spring Boot Actuator is a sub-project of the Spring Boot framework that provides production-ready features for monitoring and managing Spring Boot applications. It exposes operational information about the running application through a set of built-in endpoints. Spring Boot provides an **actuator dependency** that can be used **to monitor and manage your Spring Boot application.**

Features of Spring Boot Actuator:

- **Monitoring:** Provides insights into the application's health, performance, and resource usage.
- **Metrics:** Collects and exposes various metrics, such as JVM memory usage, HTTP request counts, and database connection pool statistics.
- **Health Checks:** Allows for checking the health status of the application and its dependencies.
- **Configuration:** Provides access to the application's configuration properties.
- **Auditing:** Tracks application events and changes.
- **Management:** Enables remote management of the application.

Advantages of Actuator Application

- It increases customer satisfaction.
- It reduces downtime.
- It boosts productivity.
- It improves Cybersecurity Management.
- It increases the conversion rate.

Spring Boot build systems are tools that automate the process of compiling, packaging, testing, and deploying Spring Boot applications.

Key Features:

Dependency Management:

Spring Boot simplifies dependency management by providing a mechanism to declare dependencies and manage their versions.

Build Automation:

Build systems automate the process of compiling source code, running tests, and packaging the application into executable formats.

Configuration:

Build systems allow for the configuration of build processes, such as specifying compiler options, test settings, and deployment targets.

Plugin Support:
Build systems support plugins that extend their functionality, enabling tasks like code analysis, code formatting, and deployment to various platforms.

Popular Build Systems for Spring Boot:

Maven:

A widely used build tool that utilizes a pom.xml file to define project configurations and dependencies.

Gradle:
A more flexible build tool that uses a Groovy-based or Kotlin-based DSL for build configuration. It is known for its performance and customization capabilities.

You can divide your project into layers like service layer, entity layer, repository layer**,,** etc. You can also divide the project into modules.

Note: *It is recommended to use Java's package naming conventions with a reverse domain name. For example,* ***com.gfg.demo****.*

- Main Application Class

It is recommended to place the Main Application class in the root package with annotations like **@SpringBootApplication** or **@ComponentScan** or **@EnableAutoConfiguration**. It allows the Spring to scan all classes in the root package and sub-packages.
two approaches that are typically used by most developers to structure their spring boot projects.

1. Structure by Feature
2. Structure by Layer
    3. **Structure 1:** By feature
    4. In this approach, all classes pertaining to a certain feature are placed in the same package. The structure by feature looks is shown in below example

    5. **Example**

```
com
 +- gfg
    +- demo
       +- MyApplication.java
       |
       +- customer
       |  +- Customer.java
       |  +- CustomerController.java
       |  +- CustomerService.java
       |  +- CustomerRepository.java
       |
       +- order
```

```
        +- Order.java
        +- OrderController.java
        +- OrderService.java
        +- OrderRepository.java
```

Structure 2: By Layer
Another way to place the classes is by layer i.e; all controllers can be placed in controllers package and services under services package and all entities under domain or model etc.

Example

```
com
 +- gfg
    +- demo
       +- MyApplication.java
       |
       +- domain
       |  +- Customer.java
       |  +- Order.java
       |
       +- controllers
       |   +- OrderController.java
       |  +- CustomerController.java
       |
       +- services
       |   +- CustomerService.java
       |   +- OrderService.java
       |
       +- repositories
           +- CustomerRepository.java
           +- OrderRepository.java
```

Spring Boot Runners

Spring Boot Runners are interfaces that allow you to execute code after the Spring Boot application has started. They're useful for tasks that need to be performed once the application context is fully initialized.

There are two main types of Spring Boot Runners:

- **CommandLineRunner:**

  This interface provides access to command-line arguments passed to the application. It has a single `run` method that accepts a `String[] args` parameter.

  ApplicationRunner:
  This interface provides access to the command-line arguments as an `ApplicationArguments` object, which offers more structured access to the arguments. It also has a single `run` method that accepts an `ApplicationArguments args` parameter.

To use a Spring Boot Runner:

- Create a class that implements either `CommandLineRunner` or `ApplicationRunner`.
- Override the `run` method to include the code you want to execute.
- Annotate the class with `@Component` or any other Spring stereotype annotation to make it a Spring bean.

```
package com.gfg.springbootrunnersexample;

import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.stereotype.Component;

@Component
public class MyApplicationRunner implements ApplicationRunner {

    @Override
    public void run(ApplicationArguments args) throws Exception {
        System.out.println("ApplicationRunner executed with options:");
        args.getOptionNames().forEach(option ->
                System.out.println(option + "=" + args.getOptionValues(option))
        );
    }
}
```

When to Use ApplicationRunner:

- **Complex Argument Parsing:** If the application requires complex parsing of command-line arguments, such as differentiating between options and non-options.
- **Structured Arguments:** When you need to access command-line arguments in a more structured way, such as when using flags and options.

```
package com.gfg.springbootrunnersexample;

import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class MyCommandLineRunner implements CommandLineRunner {

    @Override
    public void run(String... args) throws Exception {
        System.out.println("CommandLineRunner executed with arguments:");
```

```
        for (String arg : args) {
            System.out.println(arg);
        }
    }
}
```

When to Use CommandLineRunner:

- **Data Initialization:** For example, to populate the database with initial data.
- **Configuration Checks:** To verify that certain conditions or configurations are met before the application starts.
- **Task Scheduling:** To schedule or initiate background tasks that need to start with the application.

A Logger in Spring Boot is an object that generates log messages within an application. It records events, activities, and information during runtime, aiding in troubleshooting, monitoring, and maintaining application health

RESTful Web Services REST stands for **REpresentational State Transfer**.Building RESTful web services in Spring Boot involves using the Spring framework to develop APIs that follow the principles of REST . These services let clients (like web or mobile apps) interact with data using standard web methods like GET, POST, PUT, and DELETE.

Key Concepts in Simple Words:

@RestController: This is a special label you put on a class to tell Spring Boot that this class will handle web requests. It lets you send data back to the user instead of a web page. It combines two things:

@Controller: handles web requests.

@ResponseBody: sends data (not HTML) in the response.

Common HTTP Methods and How They're Used:

GET – Used to get data from the server.

In Spring Boot, use @GetMapping to handle these requests.

Example: Get a list of all users.

POST – Used to add new data.

Use @PostMapping to create something new on the server.

Example: Add a new user.

PUT – Used to update existing data.

Use @PutMapping to replace old data with new data.

Example: Update a user's information.

DELETE – Used to remove data.

Use @DeleteMapping to delete something from the server.

Example: Delete a user.

By using these annotations (@GetMapping, @PostMapping, etc.) in a class marked with @RestController, you can build a RESTful API in Spring Boot. Each URL points to a specific resource (like /users/1), and the HTTP method tells the server what action to perform. This makes your application simple, organized, and easy to scale.

Ways to handle HTTP Request

Different ways to handle HTTP requests in Spring Boot RESTful APIs:

1. HTTP Methods:

- **GET:** Used to retrieve data from the server.
- **POST:** Used to create new resources on the server.
- **PUT:** Used to update an existing resource on the server.
- **PATCH:** Used to partially update an existing resource on the server.
- **DELETE:** Used to remove a resource from the server.

2. Controller Annotations:

- **@RestController:** Marks a class as a REST controller, handling incoming requests and returning responses directly.
- **@RequestMapping:** Maps HTTP requests to specific controller methods.

- **@GetMapping, @PostMapping, @PutMapping, @PatchMapping, @DeleteMapping:** Shortcuts for @RequestMapping, specifying the HTTP method.

3. Request Parameters:

- **@RequestParam:** Extracts query parameters from the URL.
- **@PathVariable:** Extracts values from the URI path.
- **@RequestBody:** Extracts the request body (e.g., XML) and converts it to a Java object.
- **@RequestHeader:** Extracts values from the request headers.

4. Request Handling:

- **Synchronous:** Requests are handled one at a time in the order received.
- **Asynchronous:** Multiple requests can be handled simultaneously using threads.

5. Data Validation:

- **@Valid:** Triggers validation of request data using annotations from the javax.validation library.

6. Error Handling:

- **@ControllerAdvice:** Handles exceptions thrown by controllers, allowing for custom error responses.

7. REST Clients:

- **RestTemplate:** A synchronous client for making REST calls.
- **WebClient:** A non-blocking, reactive client for making REST calls.
- **RestClient:** A synchronous client with a fluent API.

8. Request Dispatching:

- **DispatcherServlet:** The central component that receives requests and routes them to the appropriate controller methods.

9. Security:

- Spring Security can be used to secure REST endpoints.

Rest Controller

In Spring Boot, @RestController is an annotation that helps you create RESTful web services easily.

Here's what it means in simple terms:

- @Controller: This is used to handle web requests and usually returns a webpage (like HTML).
- @ResponseBody: This tells Spring that the return value of a method should go directly in the response, like JSON or XML, not as a webpage.
- @RestController: This combines both @Controller and @ResponseBody. It's mainly used for building APIs.

With @RestController:

- You don't need to write @ResponseBody on every method.
- It automatically converts your data to JSON or XML.
- It sends data directly as a response, instead of showing a webpage.

So, it's a simple way to build APIs that return data.

Request Mapping

@RequestMapping in Spring Boot is an annotation used to map web requests to specific handler methods within a controller. It acts as a bridge between incoming HTTP requests and the corresponding Java methods that should handle them.

In other words @RequestMapping connects a **URL path** to a **Java method**. When someone visits that path (for example, /hello), Spring calls the method you marked with @RequestMapping.

Example:

```
@RestController
public class MyController {

    @RequestMapping("/hello")
    public String sayHello() {
        return "Hello, world!";
    }
}
```

In this example:

- When someone goes to http://localhost:8080/hello, they'll see **"Hello, world!"**

Request Body

In Spring Boot, the @RequestBody annotation is used to map the body of an HTTP request directly to a method parameter in a controller. This makes it easy to receive and process data sent in the request body, typically in formats like JSON or XML, by converting it into a Java object.

Key Details:

Purpose:

- Automatically deserializes the request body into a Java object.
- Streamlines the handling of data received from clients in your API.

How It Works:

1. **Annotation**: Place the @RequestBody annotation on a method parameter in your controller.
2. **Request Body Handling**: Spring Boot intercepts the incoming request containing a body.
3. **Conversion**: The framework uses an HttpMessageConverter to deserialize the request body into the specified Java object, guided by the Content-Type header.
4. **Parameter Binding**: The converted Java object is passed as an argument to the controller method.

This approach simplifies the interaction between client requests and server-side logic, allowing for seamless integration and processing of structured data.

Path Variable :

Path variables in Spring Boot are used to extract dynamic values from the URL path. These values are then used as parameters in controller methods. The @PathVariable annotation is used to bind the values from the URL to the method parameters.

- **How it works:**
  - In your controller method, you use the @PathVariable annotation before a method parameter.
  - The URL path is defined with placeholders enclosed in curly braces {}.
  - When a request is made, Spring maps the values from the URL to the corresponding method parameters annotated with @PathVariable.

Example

****

URL Example:

GET /users/123

Controller Example:

```
@RestController
@RequestMapping("/users")
public class UserController {

    @GetMapping("/{id}")
    public String getUserById(@PathVariable("id") String userId) {
        return "User ID: " + userId;
    }
}
```

In this example:

- `{id}` in the `@GetMapping` annotation indicates a path variable in the URL.
- The `@PathVariable("id")` annotation maps the value of the `id` placeholder in the URL to the `userId` parameter.

Sending a Request:

When a client sends a request to:

```
GET /users/123
```

The `userId` parameter in the method will be assigned the value `"123"`, and the response will be:

```
User ID: 123
```

Request Parameter

In Spring Boot, a **Request Parameter** refers to key-value pairs sent as part of the query string in an HTTP request URL. It is commonly used to send small amounts of data from a client to the server, often to filter or customize the response.

Key Features of Request Parameters:

- **URL Query Strings**: Request parameters are part of the URL after the ? symbol.
- **Annotation**: The @RequestParam annotation is used in Spring Boot to extract these parameters from the URL and bind them to method parameters.
- **Optional or Required**: Parameters can be configured as required or optional.

Example

URL Example:

For a URL like:

```
GET /search?query=spring&limit=10
```

The request parameters are:

- query with the value spring
- limit with the value 10

Controller Example:

```
@RestController
@RequestMapping("/search")
public class SearchController {

    @GetMapping
    public String search(@RequestParam("query") String query,
                         @RequestParam("limit") int limit) {
        return "Search Query: " + query + ", Limit: " + limit;
    }
}
```

Here:

- The @RequestParam("query") maps the value of the query parameter to the query method parameter.
- Similarly, @RequestParam("limit") maps the limit parameter.

Sending a Request:

When the client sends a request to:

```
GET /search?query=spring&limit=10
```

The method parameters will be populated as:

- query = "spring"
- limit = 10

The output might be:

```
Search Query: spring, Limit: 10
```

GET, POST, PUT, DELETE APIs

These are HTTP methods used in RESTful APIs to perform CRUD (Create, Read, Update, Delete) operations on resources. Spring Boot provides annotations to map these HTTP methods to specific controller methods.

GET:

Used to retrieve data from the server. GET requests should not modify data on the server.

In Spring Boot, you use the `@GetMapping` annotation to map a method to handle GET requests

POST:

Used to send data to the server to create a new resource. The data is included in the request body.

In Spring Boot, you use the `@PostMapping` annotation to map a method to handle POST requests.

PUT:

Used to update or replace an existing resource on the server. The data is included in the request body.

In Spring Boot, you use the `@PutMapping` annotation to map a method to handle PUT requests.

DELETE:

Used to remove a resource from the server.
In Spring Boot, you use the `@DeleteMapping` annotation to map a method to handle DELETE requests.
Key Differences

GET vs. POST:

GET retrieves data, while POST sends data to create or update. GET requests are visible in the URL, while POST requests have data in the request body.

PUT vs. POST:
Both can create or update, but PUT is idempotent (multiple identical requests have the same result), while POST is not. PUT usually replaces the entire resource, while POST can create a new resource or update part of an existing one.

Spring Boot- Spring Boot Configuration, Spring Boot Annotations, Spring Boot Actuator, Spring Boot Build Systems, Spring Boot Code Structure, Spring Boot Runners, Logger, BUILDING RESTFUL WEB SERVICES, Rest Controller, Request Mapping, Request Body, Path Variable, Request Parameter, GET, POST, PUT, DELETE APIs, Build Web Applications