Overview of Maven

Maven is a widely-used project management and build automation tool in the Java ecosystem. It simplifies and standardizes the build process, dependency management, and project lifecycle management, making it easier to manage complex Java projects.

Key Features:

- **Dependency Management:** Maven automatically downloads project dependencies and transitive dependencies from a central repository (Maven Central) or other configured repositories, ensuring that all required libraries are available for the build.
- **Build Automation:** Maven automates the process of compiling source code, packaging the compiled code into a distributable format (like JAR or WAR), running tests, generating documentation, and deploying the application.
- **Project Structure:** Maven enforces a standard directory structure for projects, which helps maintain consistency across different projects. The default structure includes directories for source code, test code, resources, and compiled classes.
- **POM (Project Object Model):** The heart of a Maven project is the pom.xml file, which contains configuration details such as project dependencies, plugins, build settings, and project metadata (like groupId, artifactId, and version).
- **Lifecycle Management:** Maven defines a standard build lifecycle that includes phases like validate, compile, test, package, verify, install, and deploy. Each phase represents a step in the build process, and plugins can be attached to these phases to perform specific tasks.

Advantages of Using Maven:

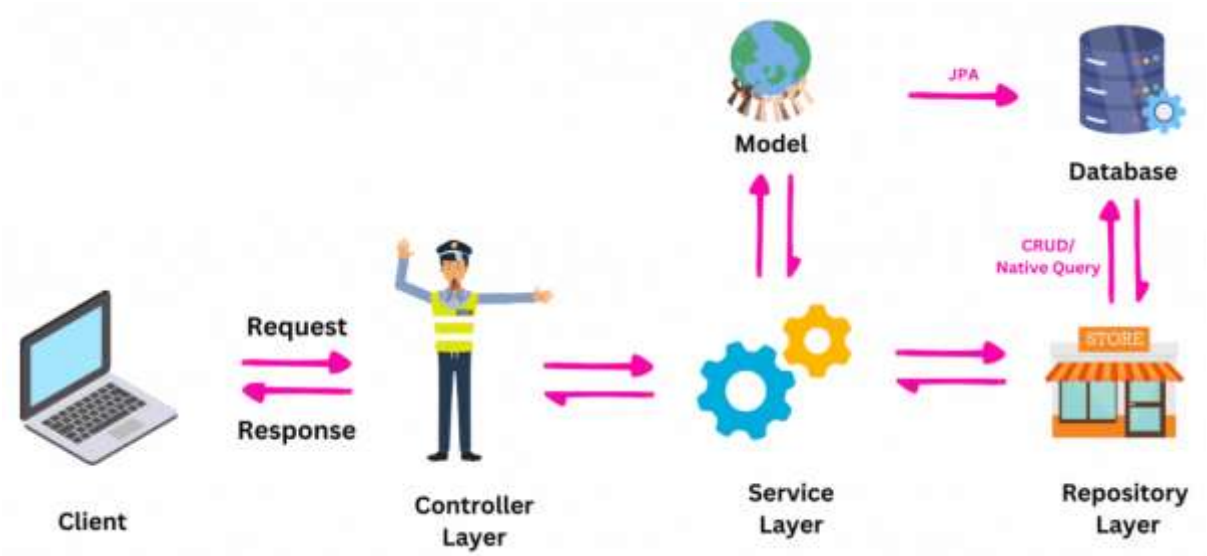
- **Consistency:** By enforcing a standard project structure and build process, Maven ensures that all projects follow a consistent approach, making it easier for teams to collaborate and for new developers to get up to speed.

- **Reusability:** Maven's dependency management system allows developers to easily reuse libraries across multiple projects without manually managing JAR files.
- **Extensibility:** Maven's plugin system allows it to be extended to support different types of projects and build processes. Numerous plugins are available for tasks like compiling code, running tests, generating reports, and deploying applications.
- **Community Support:** Maven is supported by a large community, providing a wealth of plugins, documentation, and third-party tools.

In summary, Maven is a powerful tool that streamlines the development, build, and deployment process for Java projects by automating repetitive tasks, managing dependencies, and enforcing best practices through a standard project structure.

Introduction to Spring Boot

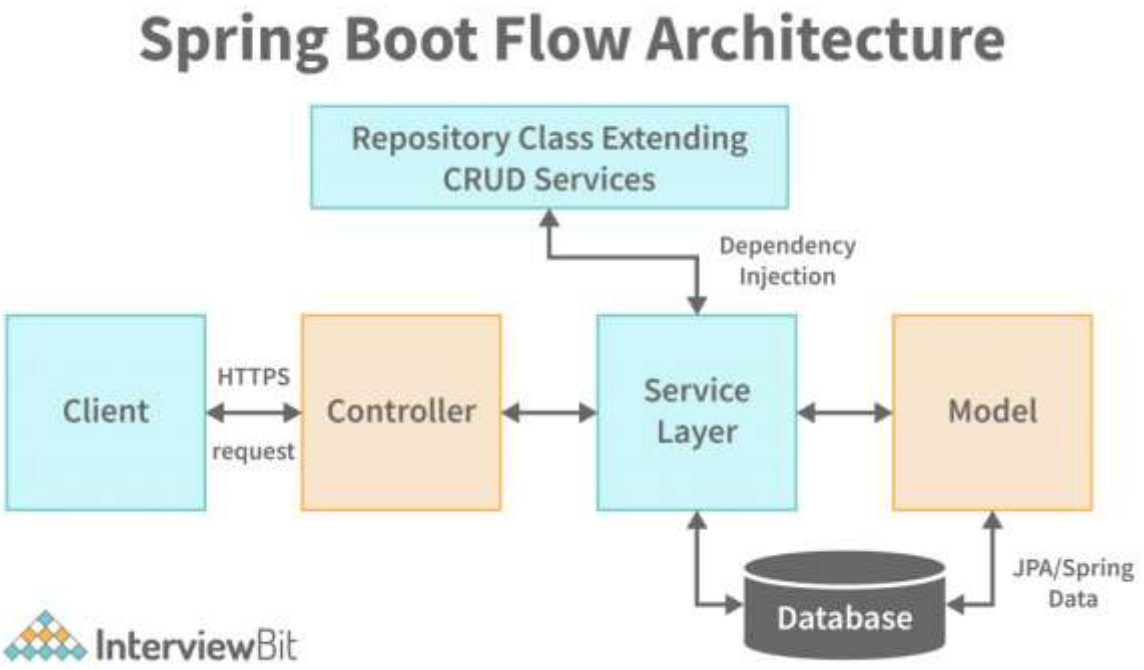
https://docs.spring.io/spring-boot/index.html



Spring Boot is a framework built on top of the Spring Framework, designed to simplify the development of Spring-based applications by providing a range of tools and features that reduce the need for boilerplate code and configuration. It allows developers to create stand-alone, production-ready applications with minimal configuration, making it easier to get started with Spring.

What is Spring Boot?

Spring Boot is an extension of the Spring Framework that takes an opinionated approach to configuration, providing defaults and conventions that help developers build applications faster. By doing so, it abstracts away much of the complexity involved in setting up and configuring Spring applications.



Key characteristics of Spring Boot include:

- **Convention over Configuration:** Spring Boot favors sensible defaults and conventions, reducing the amount of manual configuration required.
- **Embedded Servers:** Applications can run as standalone executables with embedded servers like Tomcat, Jetty, or Undertow, eliminating the need for a separate application server.
- **Production-Ready Features:** Spring Boot includes built-in support for features like health checks, metrics, and externalized configuration, which are essential for production environments.

What Does Spring Boot Do?

Spring Boot streamlines the development process by providing several key features and functionalities:

1. **Auto-Configuration:**
    - Spring Boot automatically configures your application based on the dependencies you include in your project. For mphasis, if you add a dependency for a database, Spring Boot will automatically configure a DataSource and set up a connection pool.
2. **Starter POMs:**
    - Starter POMs are Maven (or Gradle) dependency descriptors that provide a set of pre-configured dependencies for different types of applications. For instance, spring-boot-starter-web includes all the dependencies required to build a web application using Spring MVC.
3. **Embedded Servers:**
    - Spring Boot applications can be run as standalone Java applications with embedded servers like Tomcat or Jetty. This means you can package your application as a JAR file and run it directly without needing to deploy it to an external server.
4. **Production-Ready Features:**
    - Spring Boot provides several features that help make your application production-ready, such as:
        - **Actuator:** A set of tools for monitoring and managing applications, including endpoints for health checks, metrics, environment information, and more.

- **Externalized Configuration:** Spring Boot allows you to externalize configuration using .properties or .yaml files, environment variables, or command-line arguments, making it easy to adapt your application to different environments.

5. **Simplified Dependency Management:**

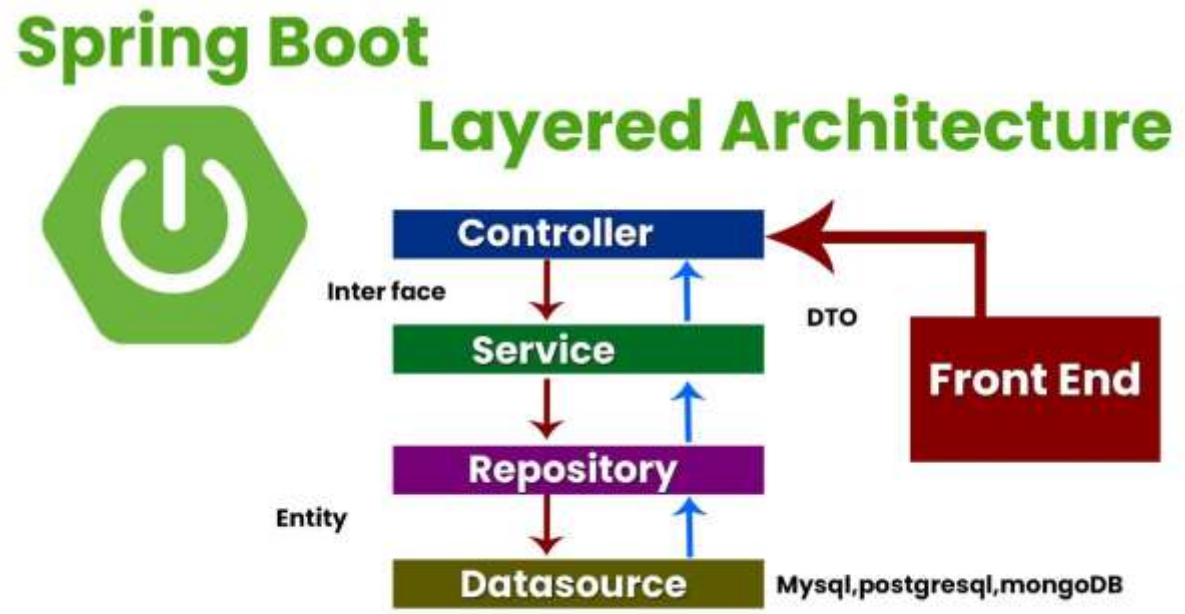
   - By using Spring Boot starter dependencies, you can easily add a set of related dependencies to your project with a single line in your pom.xml or build.gradle. This ensures compatibility between different versions of dependencies.

6. **Rapid Development:**

   - Spring Boot is designed to enable rapid application development by reducing the complexity and time required to set up and configure a Spring application. With features like embedded servers and auto-configuration, you can quickly prototype and develop applications.

Summary

Spring Boot significantly simplifies the development of Spring-based applications by providing a set of tools and conventions that reduce the need for manual configuration. By offering features like auto-configuration, embedded servers, and production-ready capabilities, Spring Boot enables developers to build stand-alone, production-ready applications quickly and efficiently.



Different APIs in Spring Boot

Spring Boot provides several important annotations and interfaces that help in configuring and controlling the behavior of your application. Three key components in this regard are @SpringBootApplication, CommandLineRunner, and ApplicationRunner. Each of these serves a distinct purpose in the lifecycle and configuration of a Spring Boot application.

1. @SpringBootApplication

@SpringBootApplication is a crucial annotation in Spring Boot that is typically placed on the main class of a Spring Boot application. This annotation is a composite of three annotations that enable a variety of configurations:

- **@Configuration:** Indicates that the class can be used by the Spring IoC container as a source of bean definitions.
- **@EnableAutoConfiguration:** Tells Spring Boot to automatically configure your application based on the dependencies present on the classpath. For instance, if spring-boot-starter-web is on the classpath, Spring Boot will automatically configure a web server.

- **@ComponentScan:** Tells Spring to scan the package (and sub-packages) where the annotated class is located for components, configurations, and services, allowing it to detect and register beans.

Mphasis:

```
package com.mphasis.demo;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.ComponentScan;

@SpringBootApplication
@ComponentScan("com.mphasis")
public class MyBoot2Application {

	public static void main(String[] args) {
		SpringApplication.run(MyBoot2Application.class, args);
	}

}
```

In this mphasis, the **@SpringBootApplication** annotation sets up the application context and triggers auto-configuration, which simplifies the initial setup process.

2. CommandLineRunner

CommandLineRunner is an interface in Spring Boot that can be implemented by a bean to run specific code after the application context has been loaded and the Spring Boot application has started. This is particularly useful for executing initialization tasks or running code at startup.

Mphasis:

```
package com.mphasis.demo;

import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;
import java.util.Arrays;

@Component
public class MyCommandLineRunner implements CommandLineRunner
{
    @Override
```

```
    public void run(String... args) throws Exception {
        System.out.println("Hello from CommandLineRunner!");
        System.out.println("Application started with
arguments: " + Arrays.toString(args));
    }
}
```

- **Explanation:** In this mphasis, the run method of CommandLineRunner will be executed as soon as the application starts. You can pass arguments to the application, which will be available in the args array.

3. ApplicationRunner

ApplicationRunner is similar to **CommandLineRunner** but provides access to the **ApplicationArguments** interface, which allows for more sophisticated processing of application arguments. It is useful when you need to work with named and non-named arguments.

Mphasis:

```
package com.mphasis.demo;
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("Hello from
ApplicationRunner!");
        System.out.println("Non-option arguments: " +
args.getNonOptionArgs());
        System.out.println("Option arguments: " +
args.getOptionNames());
        if (args.containsOption("debug")) {
            System.out.println("Debug mode is
enabled.");
        }
    }
```

}

- **Explanation:** The ApplicationRunner interface provides access to an ApplicationArguments object, which allows you to work with both named arguments (options) and unnamed arguments. This can be helpful for more complex command-line processing needs.

  mvn clean

  mvn install

  mvn spring-boot:run

  with Arguments

  mvn spring-boot:run -Dspring-boot.run.arguments=--debug,arg1,arg2

Summary

- **@SpringBootApplication:** A meta-annotation that combines **@Configuration, @EnableAutoConfiguration,** and **@ComponentScan** to simplify the setup of a Spring Boot application.
- **CommandLineRunner:** An interface that allows you to execute code immediately after the application has started, primarily used for tasks like initialization.
- **ApplicationRunner:** Similar to **CommandLineRunner** but provides more advanced handling of application arguments through the **ApplicationArguments** interface.

These APIs are essential tools in the Spring Boot ecosystem, allowing developers to configure and control the application startup process effectively.

Typical Package Structure for a Spring Boot Application

When organizing a Spring Boot application, it's important to follow a well-structured package hierarchy. This helps in maintaining clarity, scalability, and ease of navigation as your project grows. Below is a typical package structure for a Spring Boot application:

Root Package

The root package is where your main application class resides, typically annotated with @SpringBootApplication. The root package usually matches your domain name in reverse, followed by the application name.

```
src
└── main
    ├── java
    │   └── com
    │       └── mphasis
    │           └── demo
    │               ├── DemoApplication.java
    │               ├── config
    │               │   ├── AppConfig.java
    │               │   ├── AppConfigProperties.java
    │               │   ├── ProfileConfig.java
    │               ├── controller
    │               │   └── HelloController.java
    │               ├── service
    │               │   └── MyService.java
```

```
│           ├── repository
│           │   └── MyEntityRepository.java
│           ├── model
│           │   └── MyEntity.java
│           └── exception
│               ├── ResourceNotFoundException.java
│               └── GlobalExceptionHandler.java
└── resources
    ├── application.properties
    ├── application.yml
    ├── application-dev.properties
    ├── application-prod.properties
    └── application-prod.yml
```

Detailed Package Breakdown

1. **Main Application Class (DemoApplication.java)**

    - **Location:** **com.mphasis.demo**
    - **Purpose:** This is the entry point of the Spring Boot application, annotated with **@SpringBootApplication**. It triggers the auto-configuration and component scanning.

```
package com.mphasis.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

```
@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

2. Configuration Package (config)

- **Location:** **com.mphasis.demo.config**
- **Purpose:** This package holds all configuration-related classes. It may include security configurations, custom configurations using **@Configuration**, and beans definitions.

```
package com.mphasis.demo.config;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {
    @Bean
    public MyService myService() {
        return new MyService();
    }
}
```

3. Controller Package (controller)

- **Location:** **com.mphasis.demo.controller**
- **Purpose:** This package contains your REST controllers, which handle incoming HTTP requests and return responses. Controllers are typically annotated with **@RestController** or **@Controller.**

```
package com.mphasis.demo.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String sayHello() {
        return "Hello, World!";
    }
}
```

4. **Service Package (service)**
    - **Location:** **com.mphasis.demo.service**
    - **Purpose:** This package contains the service layer, where you implement the business logic. Services are typically annotated with **@Service.**

```
package com.mphasis.demo.service;

import org.springframework.stereotype.Service;

@Service
public class MyService {
    public String process() {
        return "Processing...";
    }
}
```

5. **Repository Package (repository)**

    - **Location:** **com.mphasis.demo.repository**
    - **Purpose:** This package contains the data access layer, where you interact with the database. Repositories are typically interfaces annotated with **@Repository** and extend **JpaRepository** or **CrudRepository.**

```
package com.mphasis.demo.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.mphasis.demo.model.MyEntity;
```

```
public interface MyEntityRepository extends JpaRepository<MyEntity,
Long> {
}
```

6. **Model Package (model)**
    - **Location:** com.mphasis.demo.model
    - **Purpose:** This package contains the domain models or entities representing your application's data structure. These classes are typically annotated with **@Entity** if they map to database tables.

```
package com.mphasis.demo.model;

import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class MyEntity {
    @Id
    private Long id;
    private String name;

    // Getters and setters
}
```

7. **Exception Package (exception)**

    - **Location:** com.mphasis.demo.exception
    - **Purpose:** This package holds custom exception classes and global exception handlers using **@ControllerAdvice.**

```
package com.mphasis.demo.exception;

public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String message) {
        super(message);
    }
}
```

```
package com.mphasis.demo.exception;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestControllerAdvice;

@RestControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(ResourceNotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public String handleNotFound(ResourceNotFoundException ex) {
        return ex.getMessage();
    }
}
```

Optional Packages

Depending on the complexity of your application, you may also have additional packages such as:

- **dto:** Data Transfer Objects for transferring data between layers or across the network.
- **mapper:** Classes for mapping between entities and **DTO**s.
- **util:** Utility classes with static methods or common functionalities.
- **security:** Security-related configurations and classes, like custom authentication or authorization logic.

Summary

A well-organized package structure makes your Spring Boot application easier to navigate and maintain. It separates concerns across different layers (Controller, Service, Repository, etc.), promoting modularity and reusability of code. As your project grows, maintaining a clear structure will help keep the codebase manageable and understandable for all developers involved.

Working with Properties in Spring Boot: YAML and .properties

In Spring Boot, externalized configuration is supported through various formats, with .properties and YAML files being the most commonly used. These files allow you to configure your application without changing the source code, making it easier to manage environments like development, testing, and production.

1. Using .properties Files

The .properties file format is the default option for configuring Spring Boot applications. It consists of key-value pairs, where each key represents a configuration property, and each value represents the setting for that property.

Example: application.properties

```
# Server configuration
server.port=8080

# Application-specific properties
app.name=My Spring Boot Application
app.description=This is a sample application

# Database configuration
spring.datasource.url=jdbc:mysql://localhost:3306/mphasis
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

Explanation:

- **server.port:** Configures the port on which the application will run.
- **app.name** and **app.description**: Custom application properties.
- **spring.datasource.*:** Configures the datasource for connecting to a MySQL database.

2. Using YAML Files

YAML (YAML Ain't Markup Language) is another popular format for configuration, especially because of its readability. YAML files support

hierarchical configuration, which can be more readable than the flat structure of .properties files.

Example: application.yml

```
server:
  port: 2024

app:
  name: My Spring Boot Application
  description: This is a sample application

spring:
  datasource:
    url: jdbc:mysql://localhost:3306/mphasis
    username: root
    password: root
    driver-class-name: com.mysql.cj.jdbc.Driver
```

Explanation:

- YAML files use indentation to represent nested properties.
- The structure is more intuitive and visually clear, especially when dealing with hierarchical data.

3. Loading Properties from Files

Spring Boot automatically loads properties from **application.properties** or **application.yml** files located in the src/main/resources directory or in the classpath. The properties are loaded into the Spring Environment, making them accessible throughout the application.

Logging in Spring Boot is highly customizable and can be configured through various methods, including properties files, YAML files, and programmatic configuration. Here's a guide to logging and its configuration in a Spring Boot application:

1. Default Logging Configuration

Spring Boot uses Spring Boot Starter Logging as the default logging framework, which includes:

- **Logback**: The default logging implementation.
- **Log4j2**: An alternative to Logback if you choose to use it.
- **Java Util Logging (JUL)**: Can be integrated with Logback.

2. Configuring Logging Levels

You can configure logging levels for different packages or classes in your application.properties or application.yml file.

Using application.properties:

```
# Set root logging level
logging.level.root=INFO

# Set logging level for specific packages
logging.level.org.springframework.web=DEBUG
logging.level.com.example.myapp=TRACE
```

Using application.yml:

```
logging:
  level:
    root: INFO
    org.springframework.web: DEBUG
    com.example.myapp: TRACE
```

Auto-configuration is a powerful feature of Spring Boot that simplifies the configuration of Spring applications by automatically setting up

beans and configurations based on the application's classpath and other environment settings. Here's a comprehensive overview:

What is Auto-Configuration?

Auto-configuration in Spring Boot attempts to automatically configure your Spring application based on the dependencies present on the classpath. The goal is to reduce the need for boilerplate configuration and make it easier to get up and running with sensible defaults.

How Auto-Configuration Works

1. **Classpath Scanning**: Spring Boot scans the classpath for libraries and classes. Based on what it finds, it determines what auto-configuration should be applied.
2. **Conditional Configuration**: Spring Boot uses a series of conditional annotations to decide if a particular auto-configuration should be applied. These conditions check for the presence of certain classes, properties, or beans.
3. **@Configuration** Classes**:** Auto-configuration is implemented using @Configuration classes that are part of Spring Boot's spring-boot-autoconfigure module. These classes define beans and settings that should be applied if certain conditions are met.
4. **Spring Factories**: The auto-configuration classes are listed in META-INF/spring.factories files, which Spring Boot reads to know which configurations to apply.

Key Annotations

- **@EnableAutoConfiguration**: This annotation is used to enable auto-configuration. It is typically included in the **@SpringBootApplication** annotation, which is a convenience annotation that combines **@EnableAutoConfiguration**, **@Configuration, and @ComponentScan.**
- **@ConditionalOnClass:** This annotation is used to apply a configuration only if a specific class is present on the classpath.

- **@ConditionalOnMissingBean**: This annotation ensures that a bean is only created if no other bean of the same type is already defined.
- **@ConditionalOnProperty**: This annotation applies a configuration based on the presence or value of a property.
- **@ConditionalOnBean**: This annotation applies a configuration if a specific bean is already present in the application context.

Examples of Auto-Configuration

Example 1: DataSource Auto-Configuration

Spring Boot can automatically configure a DataSource bean if the application has a database dependency (like **H2, MySQL**) on the classpath and the necessary properties are set in **application.properties or application.yml.**

```
# Example properties for auto-configuring DataSource
spring.datasource.url=jdbc:mysql://localhost:3306/mphasis
spring.datasource.username=root
spring.datasource.password=root
```

Example 2: Web MVC Auto-Configuration

When using Spring Boot's web starter, it **auto-configures** a **DispatcherServlet, view resolvers**, and **other web components** based on the classpath and properties.

Example 3: Security Auto-Configuration

If you include Spring Security on the classpath, Spring Boot auto-configures basic security settings like authentication and authorization with default settings.

Customizing Auto-Configuration

1. **Exclude Auto-Configuration Classes**: If you want to disable certain auto-configuration classes, you can use the exclude

attribute of **@SpringBootApplication or @EnableAutoConfiguration.**

```
@SpringBootApplication(exclude = {DataSourceAutoConfiguration.class})
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

2. **Override Default Configuration**: You can define your own beans or configurations that override the defaults provided by auto-configuration. Spring Boot will use your custom beans if they match the expected types.
3. **Conditional Configuration**: Use conditional annotations to apply configurations based on specific conditions, such as properties or classpath availability.

How to Discover Auto-Configuration

- **Actuator Endpoints**: Use the **/actuator/autoconfig** endpoint provided by Spring Boot Actuator to see a list of all auto-configured classes and their status.
- **spring-boot-autoconfigure**: Review the spring-boot-autoconfigure module source code to understand the available auto-configurations.

Summary

Spring Boot's auto-configuration feature significantly reduces the amount of manual configuration required for common scenarios by automatically setting up beans and configurations based on the classpath and environment. It works through a combination of conditional annotations and predefined configurations, with options to customize or exclude specific auto-configurations as needed.

Customizing Spring Boot applications allows you to tailor the default behaviors and configurations to meet your specific needs. This can

involve configuring various aspects of the application, such as how beans are created, modifying default settings, or integrating custom components. Here's a guide to customizing Spring Boot applications:

1. Customizing Bean Definitions

You can define and customize beans in your Spring Boot application by using Java configuration classes.

Example: Customizing a DataSource Bean

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

@Configuration
public class DataSourceConfig {

    @Bean
    public DriverManagerDataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/mydb");
        dataSource.setUsername("root");
        dataSource.setPassword("secret");
        return dataSource;
    }
}
```

2. Overriding Default Auto-Configuration

You can override or exclude specific auto-configurations provided by Spring Boot.

Example: Excluding Auto-Configuration

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration;

@SpringBootApplication(exclude = {DataSourceAutoConfiguration.class})
```

```
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

Example: Customizing Auto-Configuration

To customize the default behavior provided by auto-configuration, define your own beans that match the type expected by the auto-configuration.

```
import
org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfigurati
on;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class CustomWebMvcConfig {

    @Bean
    public WebMvcAutoConfiguration webMvcAutoConfiguration() {
        // Custom configuration or bean overrides
        return new WebMvcAutoConfiguration();
    }
}
```

3. Custom Properties

Spring Boot applications use application.properties or application.yml files to externalize configuration. You can customize these settings to adjust application behavior.

Example: Custom Properties File

```
# Custom server port
server.port=9090

# Custom logging level
logging.level.com.example=DEBUG
```

```
# Custom application properties
app.custom-property=value
```

4. Customizing Spring Boot's Banner

You can customize or disable the startup banner shown when the application starts.

Example: Custom Banner

Create a banner.txt file in src/main/resources:

```
  ____              _            _
 | __ )  __ _ _ __ | | _____ _ __ | |_
 |  _ \ / _` / __| |/ / _ \ '_ \| __|
 | |_) | (_| \__ \   <  __/ | | | |_
 |____/ \__,_|___/_|\_\___|_| |_|\__|
```

To disable the banner:

spring.main.banner-mode=off

5. Customizing Error Handling

Customize error handling by defining your own ErrorController or using **@ControllerAdvice** for global exception handling.

Example: Custom Error Page

```
import org.springframework.boot.web.servlet.error.ErrorController;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class CustomErrorController implements ErrorController {

    @RequestMapping("/error")
    public String handleError() {
        // Return a custom error view
        return "customError";
    }
```

```
    @Override
    public String getErrorPath() {
        return "/error";
    }
}
```

Example: Global Exception Handling

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;

@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleException(Exception e) {
        return new ResponseEntity<>("Custom error message",
HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

6. Customizing Security

If using Spring Security, you can customize security settings by creating a custom security configuration class.

Example: Custom Security Configuration

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.builders.AuthenticationM
anagerBuilder;
import
org.springframework.security.config.annotation.web.configuration.WebSecurit
yConfigurerAdapter;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
```

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
            .antMatchers("/public/**").permitAll()
            .anyRequest().authenticated()
            .and()
            .formLogin();
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws
Exception {
        auth
            .inMemoryAuthentication()

.withUser("user").password(passwordEncoder().encode("password")).roles("US
ER");
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

7. Customizing Actuator Endpoints

Spring Boot Actuator provides endpoints to monitor and manage your application. You can customize these endpoints and their exposure.

Example: Customizing Actuator Endpoints

```
management.endpoints.web.exposure.include=health,info
management.endpoint.health.show-details=always
```

Example: Custom Actuator Endpoint

```
import org.springframework.boot.actuate.endpoint.annotation.Endpoint;
import org.springframework.boot.actuate.endpoint.annotation.ReadOperation;
```

```
import org.springframework.stereotype.Component;

@Component
@Endpoint(id = "custom")
public class CustomEndpoint {

   @ReadOperation
   public String customEndpoint() {
      return "Custom endpoint response";
   }
}
```

Summary

Customizing a Spring Boot application involves:

- **Bean Definitions**: Create or override beans to customize configurations.
- **Auto-Configuration**: Exclude or customize auto-configured components.
- **Properties**: Adjust application behavior through application.properties or application.yml.
- **Banner**: Customize or disable the startup banner.
- **Error Handling**: Define custom error pages or global exception handling.
- **Security**: Configure custom security settings.
- **Actuator Endpoints**: Customize or create new actuator endpoints.

These customization options allow you to tailor Spring Boot to meet the specific needs of your application, ensuring it behaves exactly as required.