Java Spring Boot

KienNCT3 - Mentor HaiNV21

Contents

1. What is Spring Boot
2. Advantages of Spring Boot
3. Create first project
4. File construction in project
5. Popular annotations in Spring Boot
6. First project with “Hello World!”
7. Externalized Configuration

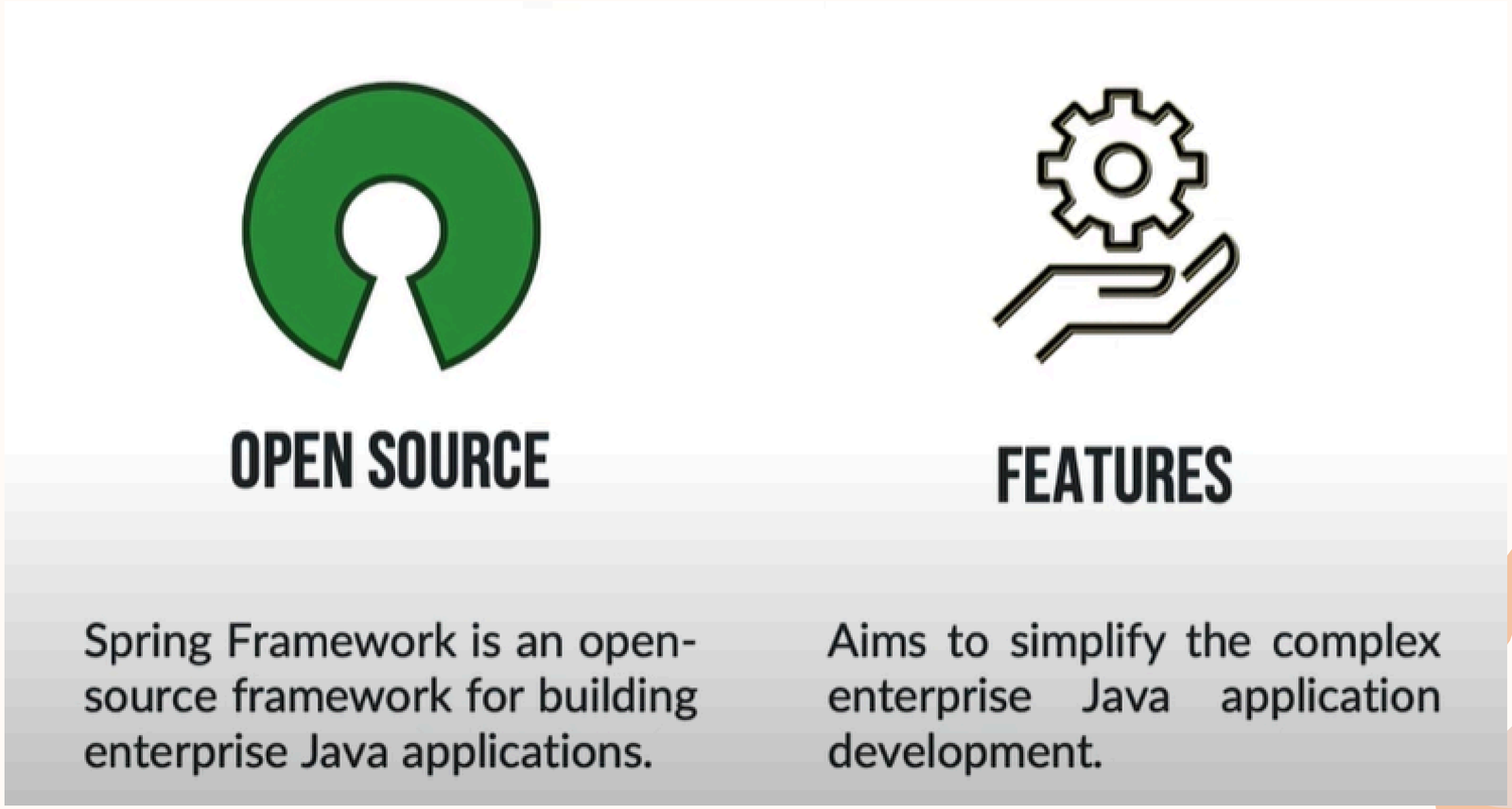
Externalized Configuration

1. Definition
2. Configuration files that default Spring Boot looks for to config
3. Some approaches to load and access the property value
4. Approach to load properties value multiple files
5. Approach to fetch and provide the default value for the placeholder
6. Some ways to handle missing property file

WHAT IS SPRING BOOT?

WHAT IS SPRING FRAMEWORK?

- Spring Framework is an open-source framework for building enterprise Java applications.
- Its aim to simplify the complex enterprise Java application development by providing some features like AOP, Dependency Injection, etc.



SO WHAT IS SPRING BOOT?

-Spring boot is an open-source, Sping-based framework developed , it was created to remove complicated feature of Spring (not request XML file) and make building applications with Spring Framework simpler and faster.

- It provides automatic configuration and dependency management to reduce complexity in building applications.

Spring Boot

=

Spring Framework + Embedded HTTP Servers (Tomcat, Jetty, …) - XML <bean> Configuration or @Configuration

ADVANTAGES OF SPRING BOOT

1) Standalone app

- Spring Boot allows independent application development without dependence on complex containers such as Tomcat.
- This provides flexibility and convenience in deploying and managing applications across many different environments.

2) Embded server

Spring Boot integrates embedded servers such as Tomcat, Jetty...

3) Starters

Spring Boot Starters helps increase productivity by automatically adding necessary dependencies to projects

4) Auto configuration

- Spring Boot uses an Auto Configuration mechanism to reduce the work required to configure the application.
- Default configurations are applied automatically, reducing the code needed to write and increasing project consistency.

5) No XML configuration

- Spring Boot encourages the use of code configuration (Java Configuration) instead of XML, which reduces complexity and makes the source code easier to read and maintain.
- Not having XML also helps avoid common mistakes related to configuration management.

CREATE FIRST PROJECT WITH SPRING BOOT

New Project

Empty Project

Generators

- Maven Archetype
- Jakarta EE
- Spring Initializr
- JavaFX
- Quarkus
- Micronaut
- Ktor
- Compose for Desktop
- HTML
- React
- Express
- Angular CLI
- Vue.js
- Vite

Server URL: start.spring.io

Name: DemoSpringBoot

Location: D:\Fsoft

Project will be created in: D:\Fsoft\DemoSpringBoot

☐ Create Git repository

Language: Java | Kotlin | Groovy

Type: Gradle - Groovy | Gradle - Kotlin | Maven

Group: org.example

Artifact: DemoSpringBoot

Package name: org.example.demospringboot

JDK: 17 Oracle OpenJDK version 17.0.9

Java: 17

Packaging: Jar | War

Next | Cancel

Spring Boot: 3.2.1

Dependencies:

Search

- Developer Tools
  - GraalVM Native Support
  - GraphQL DGS Code Generation
  - [x] Spring Boot DevTools
  - [x] Lombok
  - Spring Configuration Processor
  - Docker Compose Support
  - Spring Modulith
- Web
  - [x] Spring Web
  - Spring Reactive Web
  - Spring for GraphQL
  - Rest Repositories
  - Spring Session
  - Rest Repositories HAL Explorer
  - Spring HATEOAS
  - Spring Web Services
  - Jersey
  - Vaadin

Thymeleaf

A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

Guide ↗

Added dependencies:

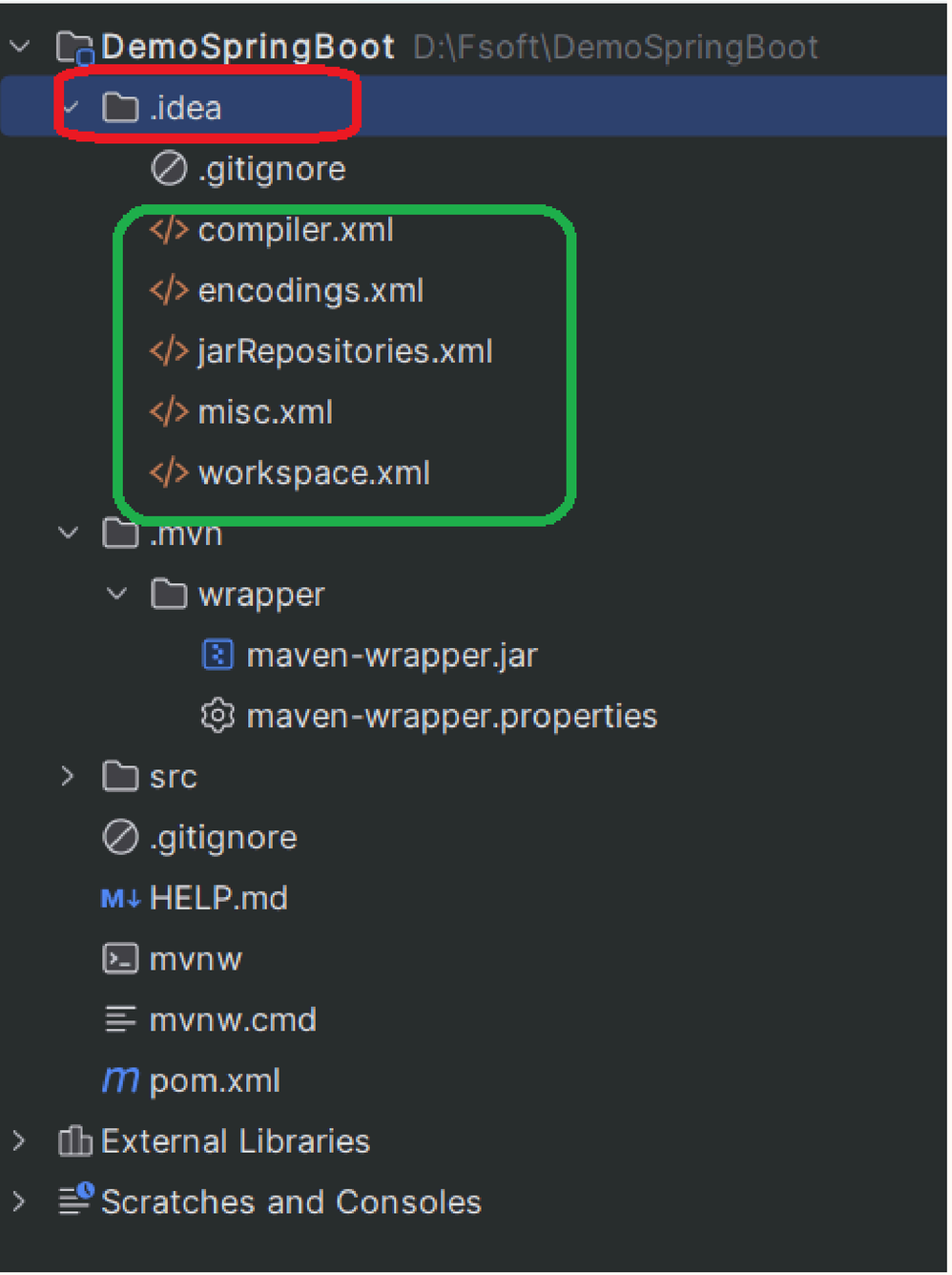
- Lombok
- Spring Boot DevTools
- Spring Web
- Thymeleaf

Previous | Create | Cancel

Project

- DemoSpringBoot D:\Fsoft\DemoSpringBoot
  - .idea
    - .gitignore
    - compiler.xml
    - encodings.xml
    - jarRepositories.xml
    - misc.xml
    - workspace.xml
  - .mvn
    - wrapper
      - maven-wrapper.jar
      - maven-wrapper.properties
  - src
    - main
    - test
  - .gitignore
  - HELP.md
  - mvnw
  - mvnw.cmd
  - pom.xml
- External Libraries
- Scratches and Consoles

File construction of a Spring boot project



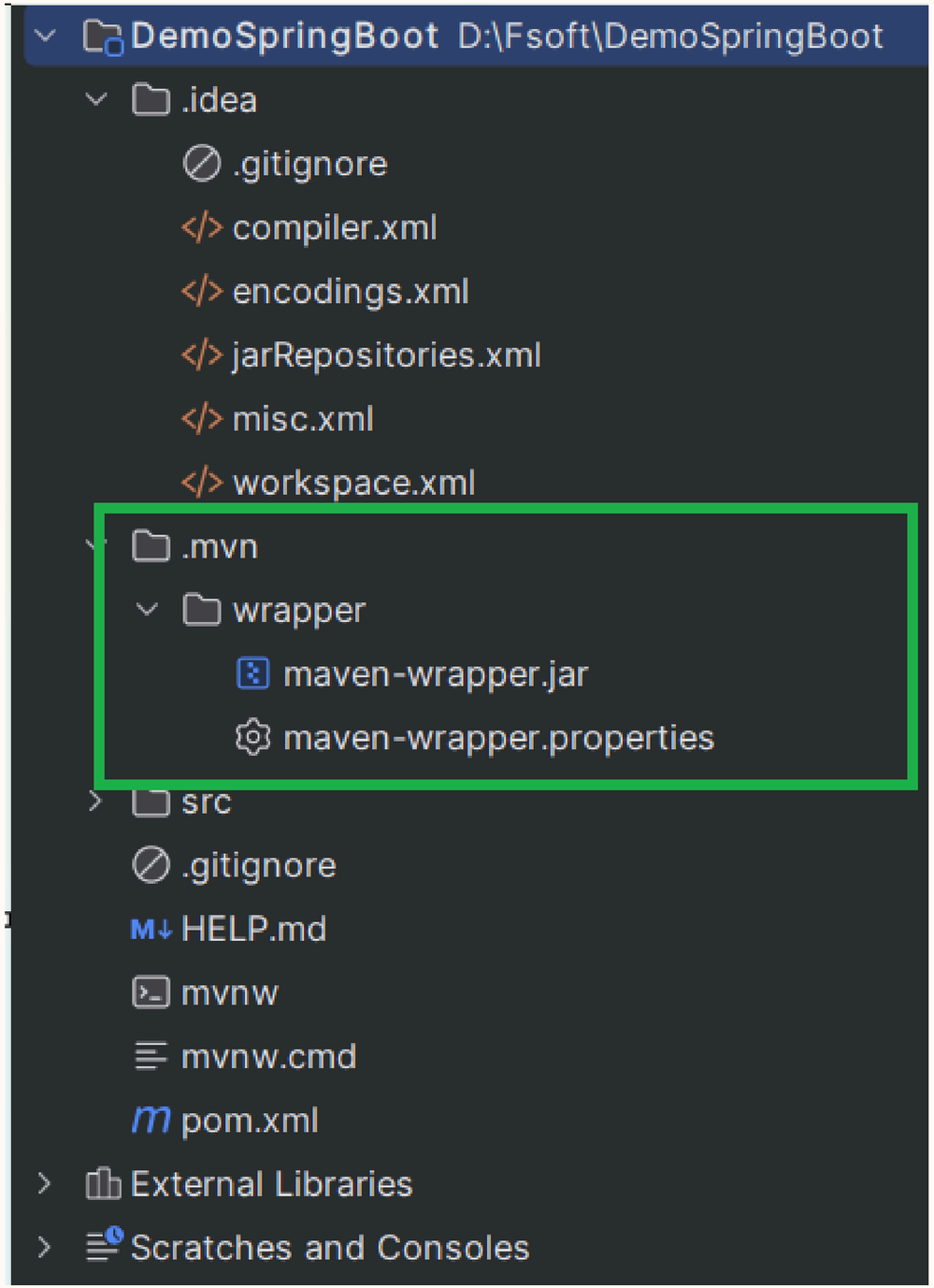
- idea: This is the folder containing the project configurations for IntelliJ

+ idea/compiler.xml: Contains settings related to the compilation process

+ idea/encodings.xml
+ idea/jarRepositories.xml

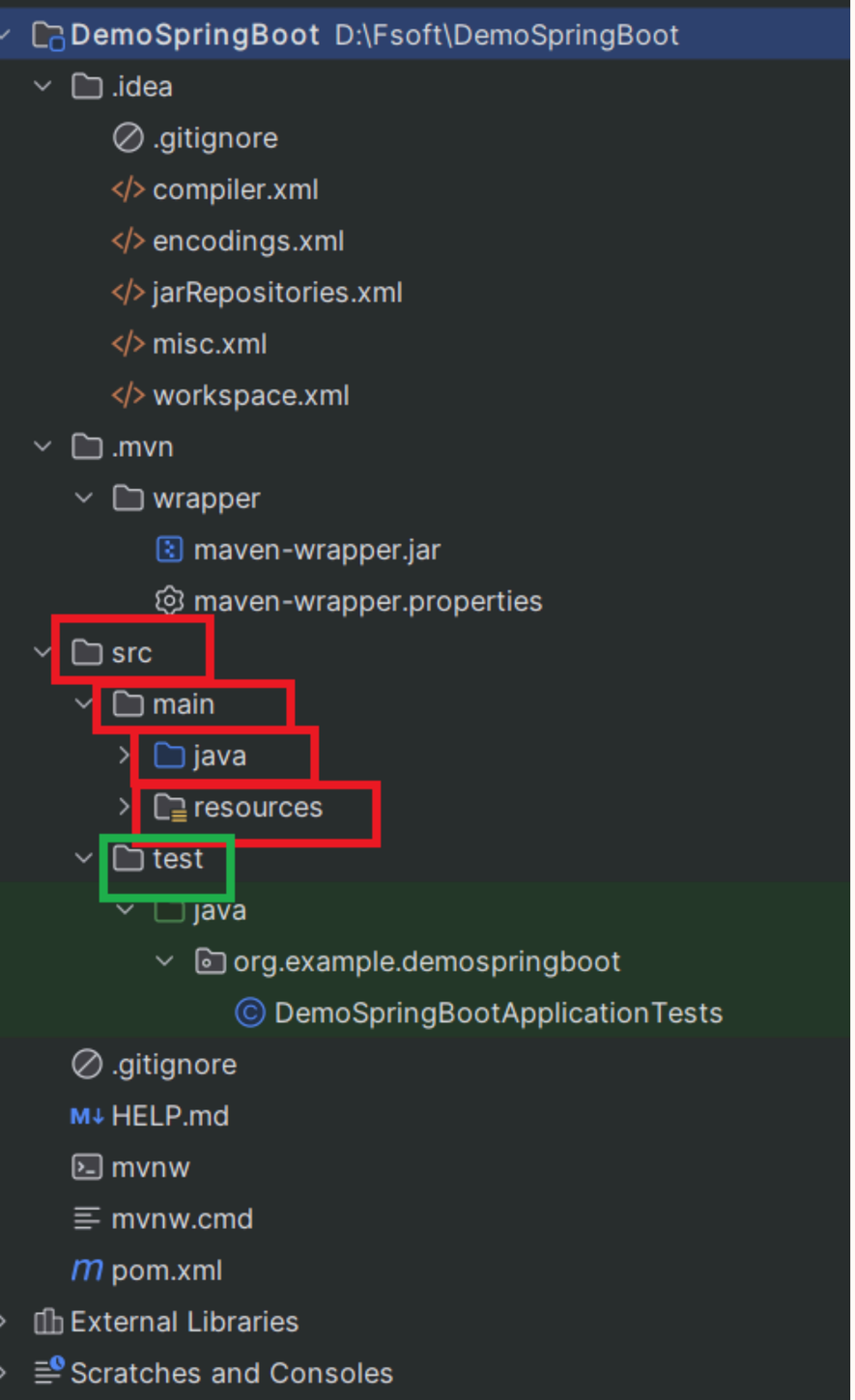
+ idea/misc.xml: Contains other project configurations that are not part of other sections such as settings, code inspection rules, and code formatting settings.

+ idea/workspace.xml: contain detail config of application in IDE include enviroment, open tabs, etc



- mvn: This directory contains configuration files for Maven.

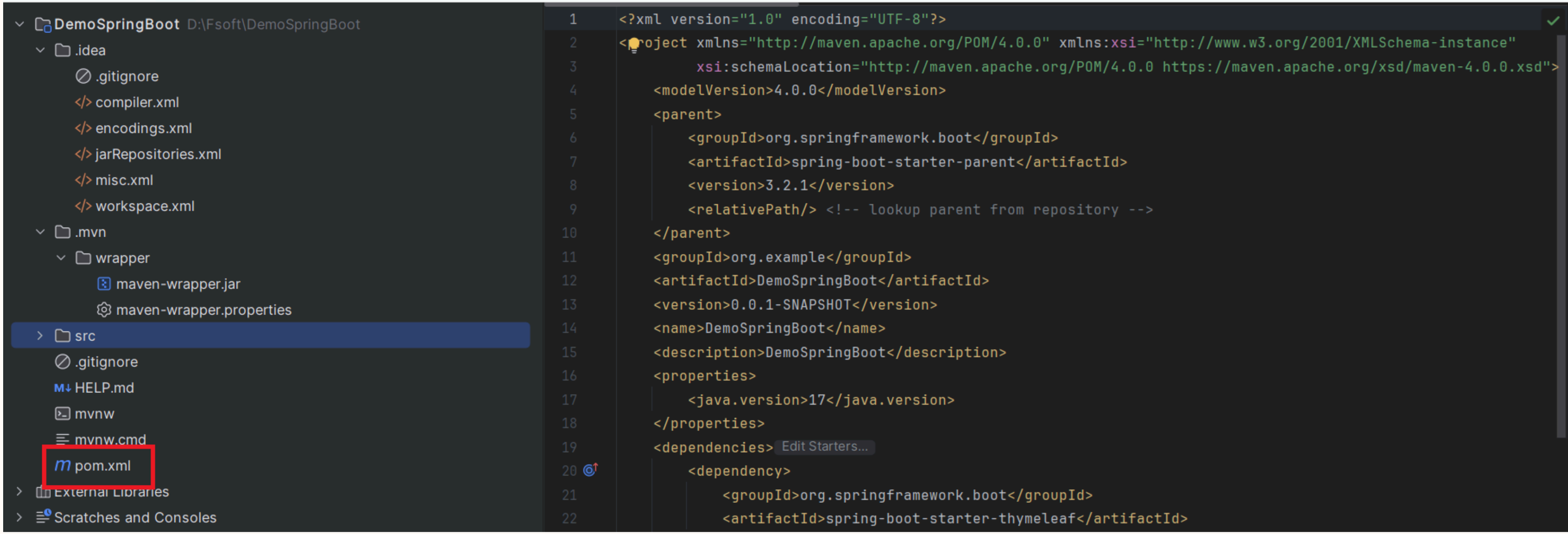
+ mvn/wrapper/maven-wrapper.properties: Maven Wrapper configuration, helps ensure each project user uses the same Maven version.



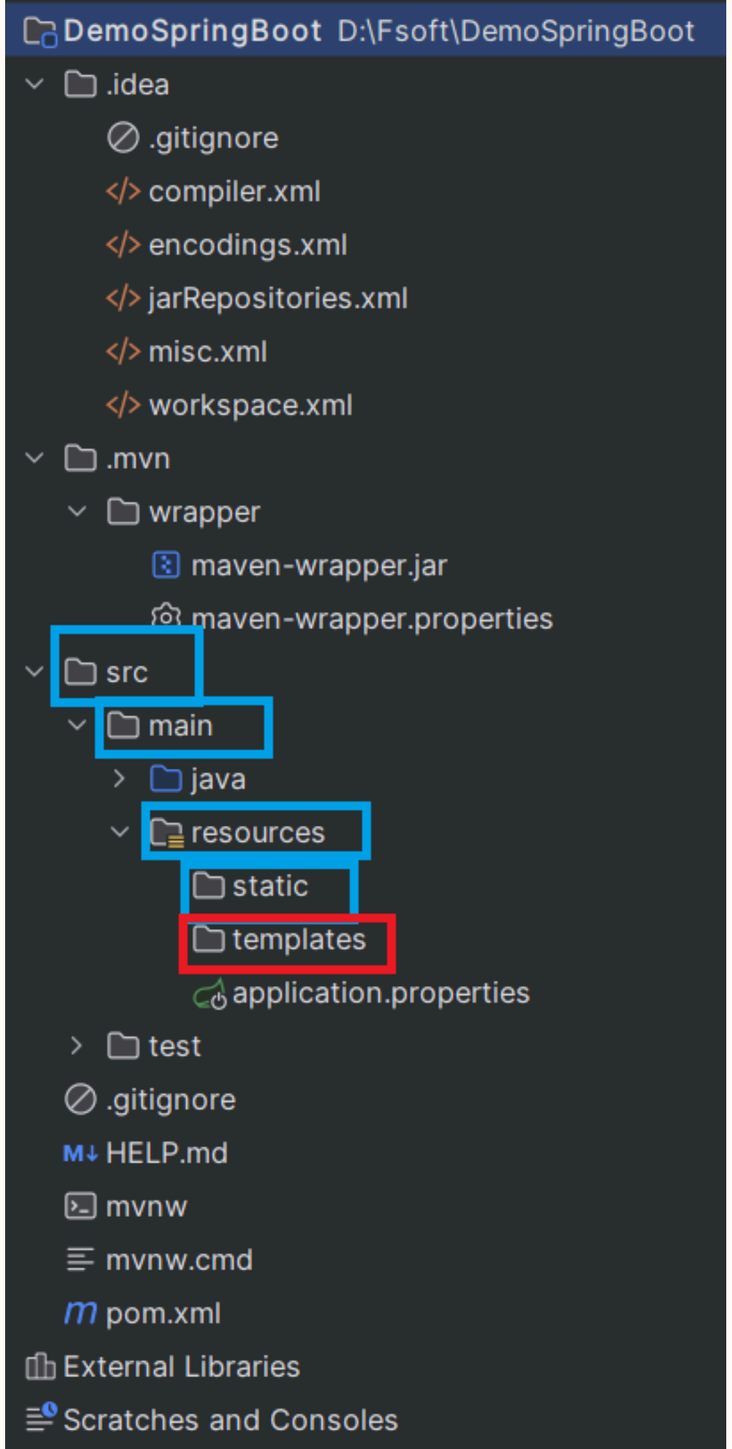
- src/main/java: use to store Java code of application

- src/main/rescources: contains resources such as configuration files, templates, images, and other resources. Important files included:
+ application.properties or application.yml: Main configuration file of application

- src/test: contain tests for main code

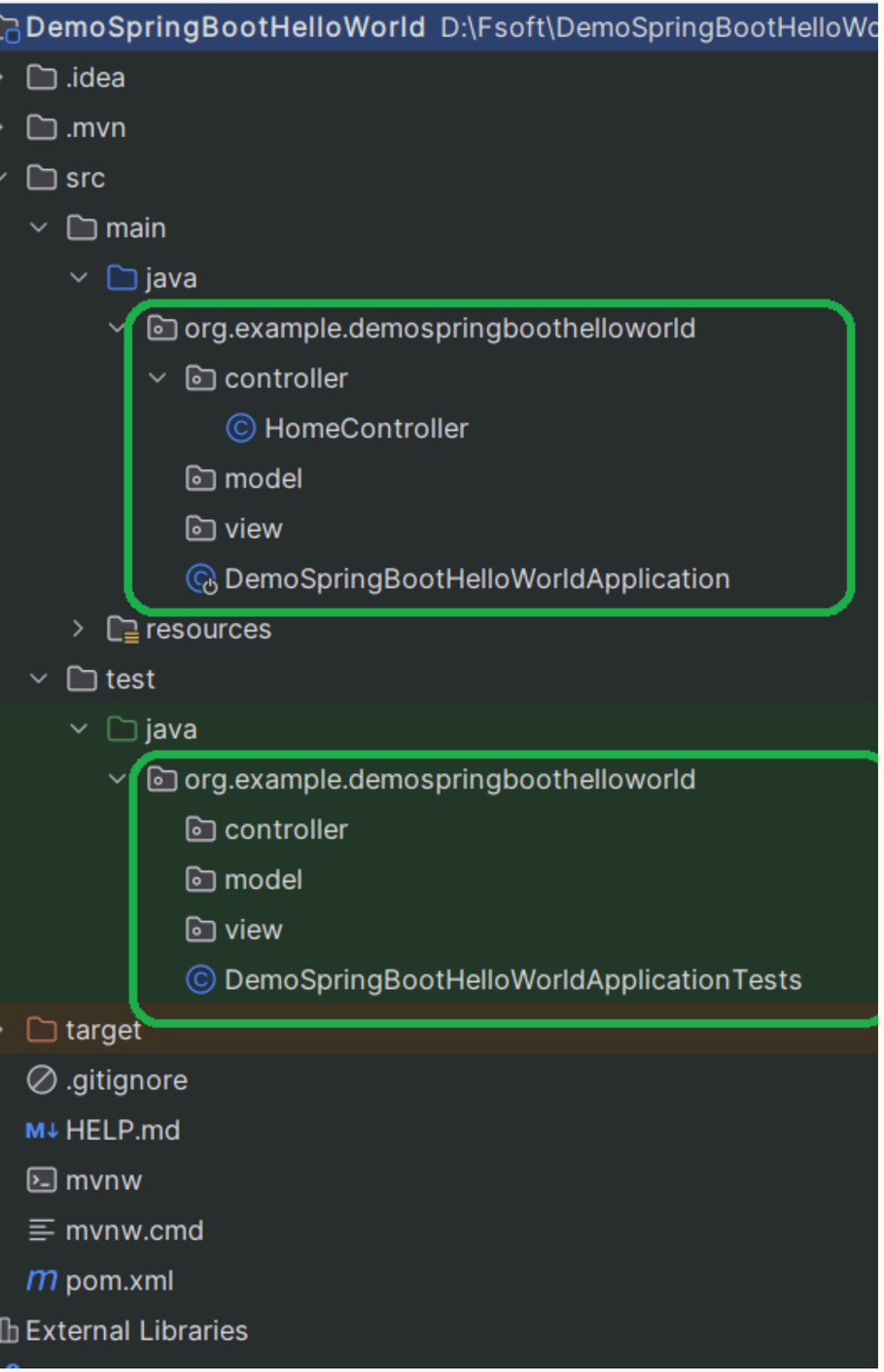


pom.xml: Maven configuration file to manage project dependencies and settings.



- src/main/resources/static: contains static resources such as CSS, JavaScript, and images.

- src/main/resources/templates: contains templates for websites or other template frameworks.



- src/main/java/your_pakage: containing the application's Java source code, organized by your package.

- src/test/java/your_package: containing tests of application, organized by your package.

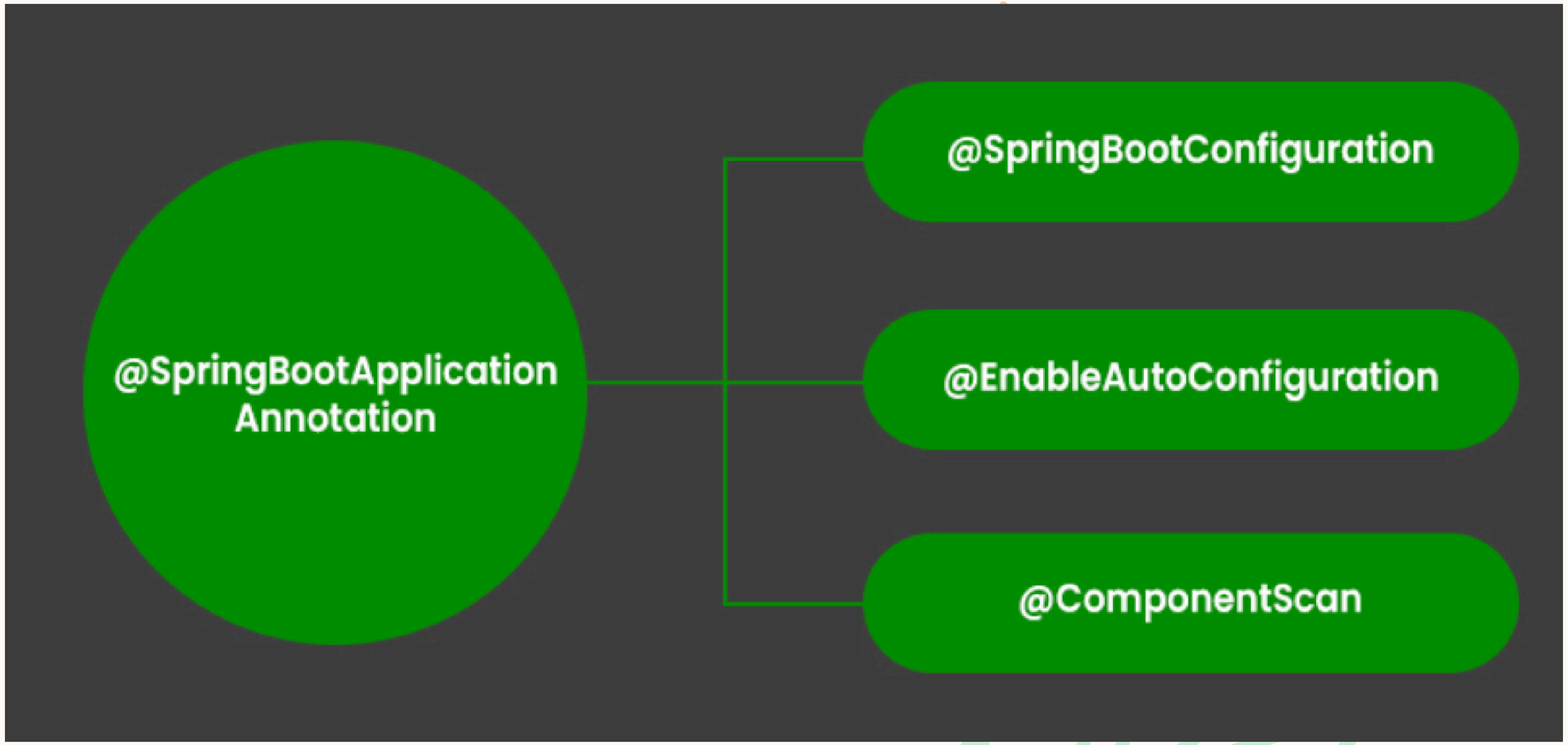
Spring Boot Annotations

1) @SpringBootApplication: marks the class as a main class of the Spring Boot application.

```
@SpringBootApplication
public class DemoSpringBootHelloWorldApplication {

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

1) @SpringBootApplication: marks the class as a main class of the Spring Boot application. It encapsulates @SpringBootConfiguration, @EnableAutoConfiguration, and @ComponentScan annotations with their default attributes.

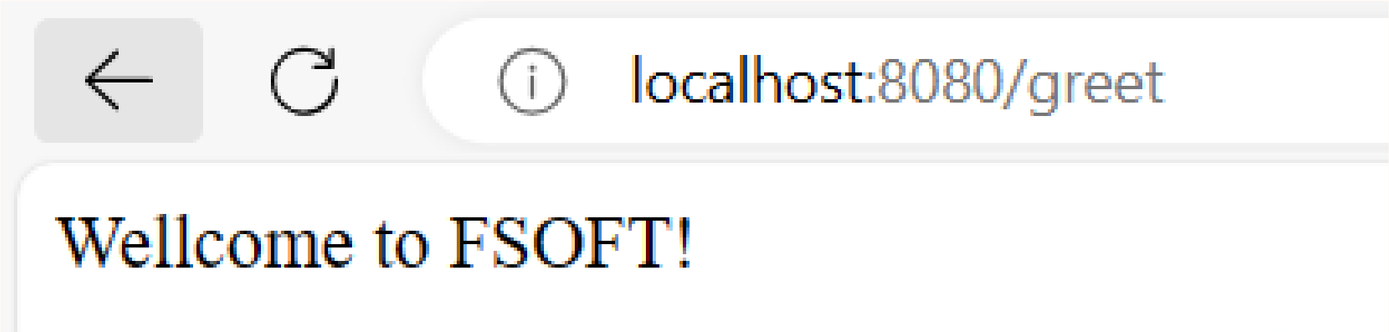


2) @Controller: use to mark the class as a SPRING MVC Controller, use to handle HTTP requests and return view.

```
@Controller
public class HomeController {
    @GetMapping("/hello")
    public String hello() {
        //Process request and return view "hello"
        return "hello";
    }
}
```

3) @RestController: It combines two other annotation @Controller and @ResponseBody. It is used to create web services in a Spring application.

```java
@RestController
public class HomeController {
    @GetMapping("/greet")
    public ResponseEntity<String> greet() {
        String message = "Wellcome to FSOFT!";
        return new ResponseEntity<>(message, HttpStatus.OK);
    }
}
```



4) @RequestMapping: Use to mapping HTTP request to controller or method to handle request.

```java
@RestController
@RequestMapping("/wellcomFSOFT")
public class HomeController {
    @GetMapping("/hello")
    public String hello() {
        return "hello world";
    }
}
```

5) @GetMapping: is a child annotation of RequestMapping and helps shorten the syntax

```
@RestController
public class HomeController {
    @GetMapping("/hello")
    public String hello() {
        //process GET request to /hello and then return to view "Nguyen Cao Trung Kien"
        return "Nguyen Cao Trung Kien";
    }
}
```

6) @Component:

- Used to mark a class as a Spring component
- Spring boot will manage and automatically create beans from the class marked by @Component when the application is started.

```
@Component
public class MyConfig {
    no usages
    public void display() {
        System.out.println("Wellcome to FSOFT!");
    }
}
```

7) @Autowired:

- Used to automatically uinject dependencies into beans.

```java
@RestController
public class HomeController {
    @Autowired
    private MyConfig myMethod;

    @GetMapping("/api")
    public String getMess() {
        return myMethod.display();
    }

}
```

7) @Service:
- Mark a class as a Service bean in the application.
- Usually used to contain business logic.

8) @Repository:
- Mark a class as a Spring Data Repository.
- Provides common methods for interacting with the database.

FIRST PROJECT WITH "HELLO WORLD"

HOW TO RUN PROJECT WITH SPRING BOOT FRAMEWORK

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

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

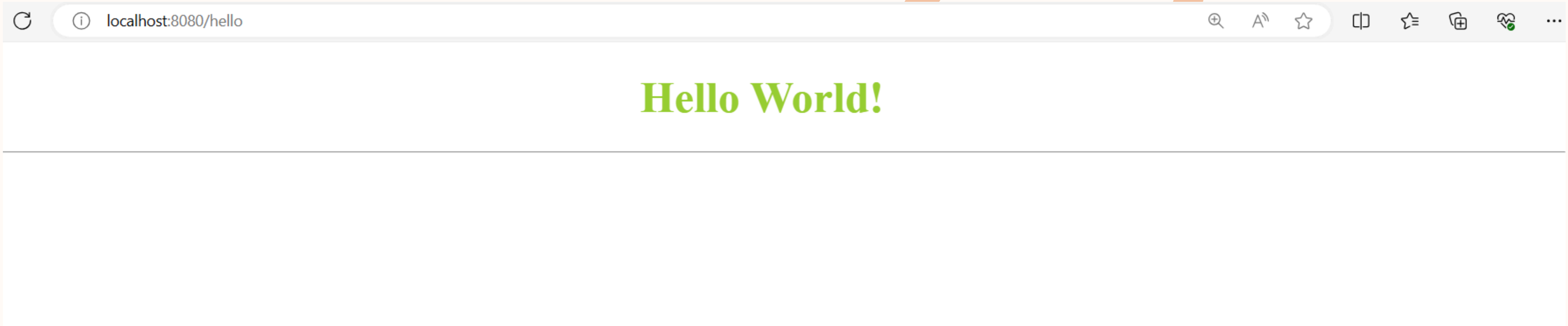
Use annotation @SpringBootApplication on main class, then call SpringApplication.run(.class, args) to run project.

```java
@RestController
public class HomeController {
    @GetMapping("hello")
    public ModelAndView index() {
        return new ModelAndView( viewName: "index");
    }
}
```

@RestController: This annotation indicated that this class was a controller, where handle requests from user.

@GetMapping("hello"): This annotation indicated when user access to address "/hello", the system will return file index.html through Model and View class

View in localhost:

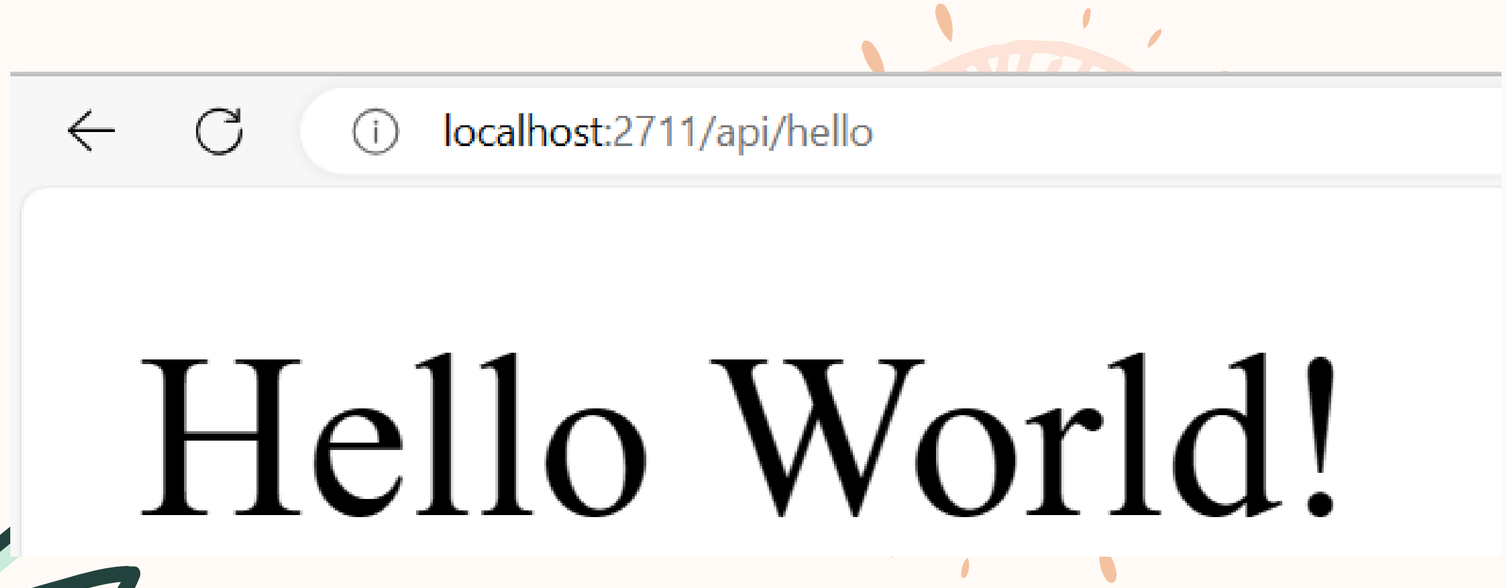


@RestController: This annotation indicated that this class was a controller, where give requests from user.

@GetMapping("hello"): This annotation indicated when user access to address "/hello", the system will return view "Hello World!"

```
@RestController
public class HomeController {
    @GetMapping("hello")
    public String hello() {
        return "Hello World!";
    }
}
```

View in localhost:



Externalized Configuration in Spring Boot

DEFINITION:

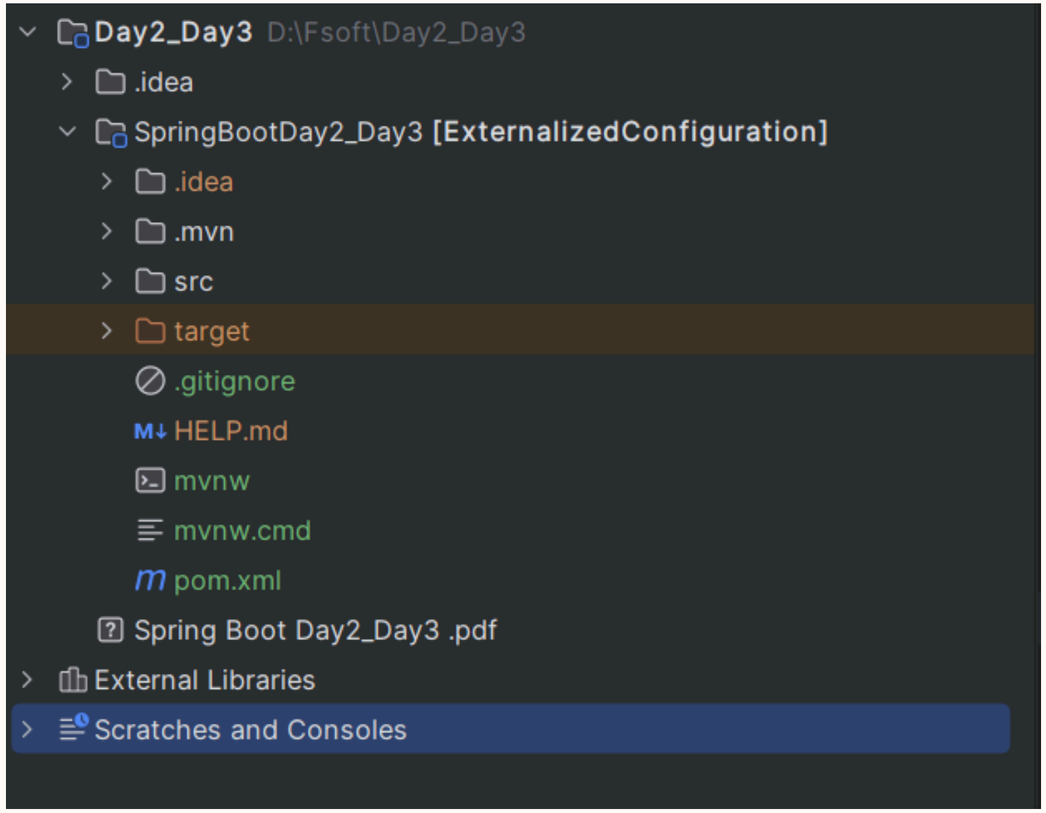
- Externalized Configuration is an important feature of Spring Boot, helping manage application configuration flexibly from outside the source code.
- Instead of locking configurations in the source code, we can leverage external configuration files like application.property, application.yml, or even other data sources like Environment Variables.

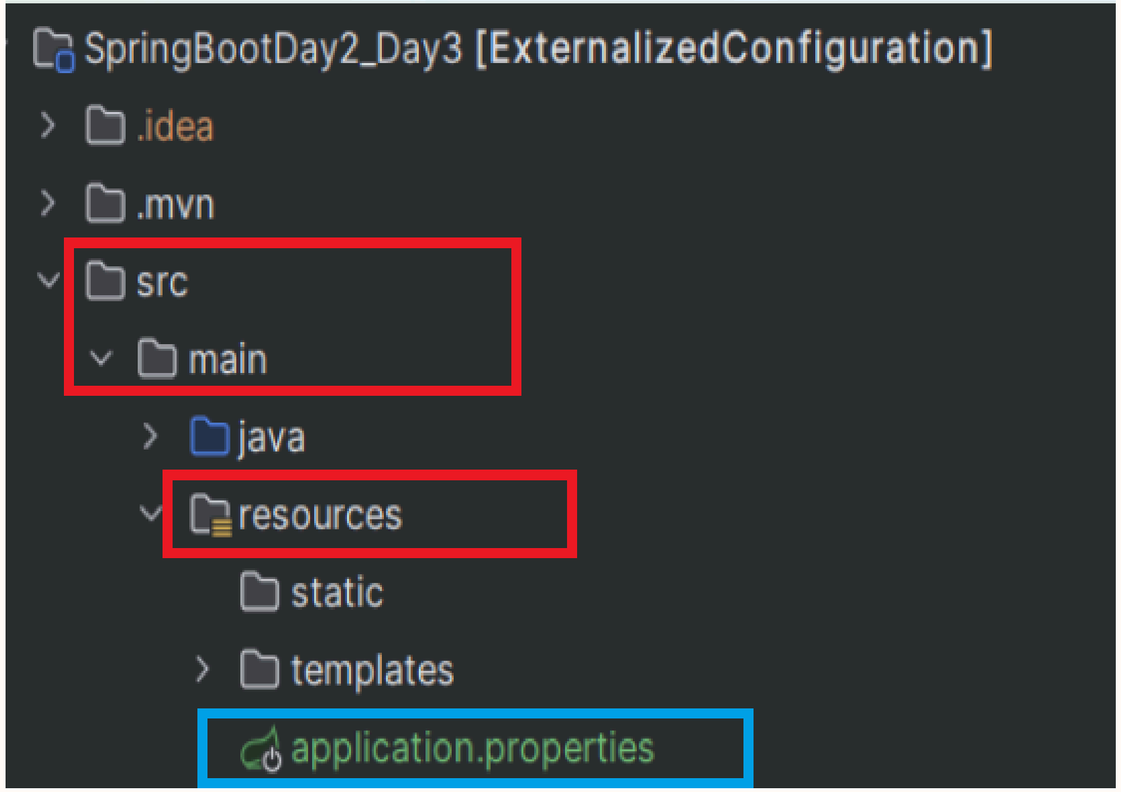
FOR EXAMPLE:

application.properties

```
1    server.port = 3030
```

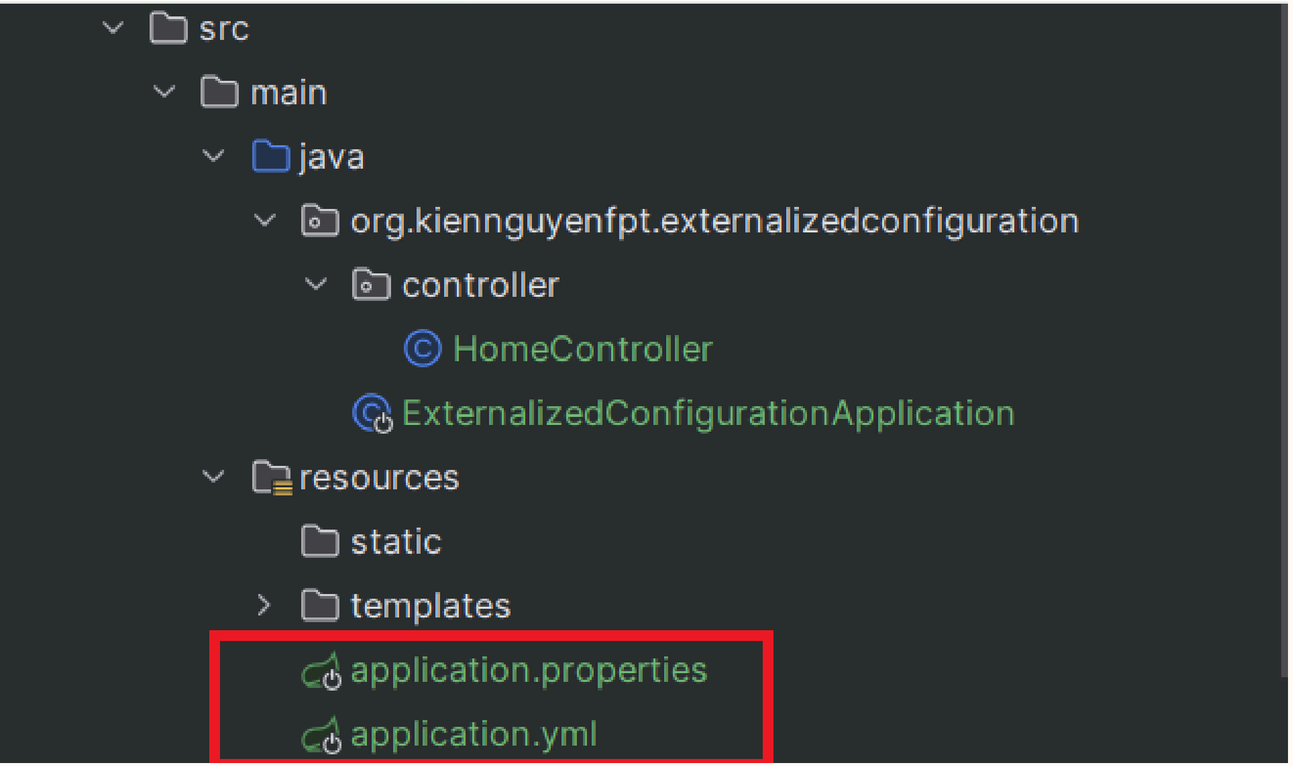
- By default, Spring Boot load file application.properties, application.yml to configure

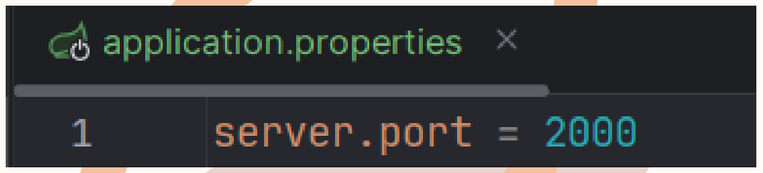


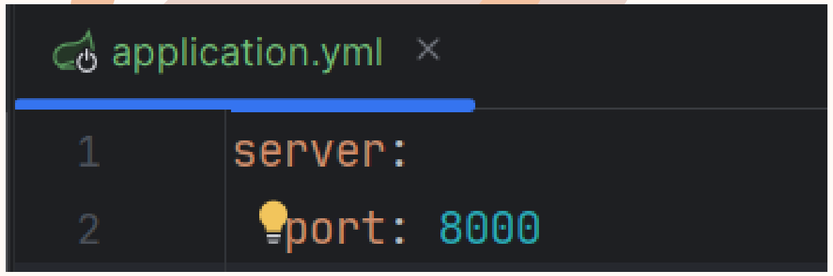


Default Spring Boot Load File:

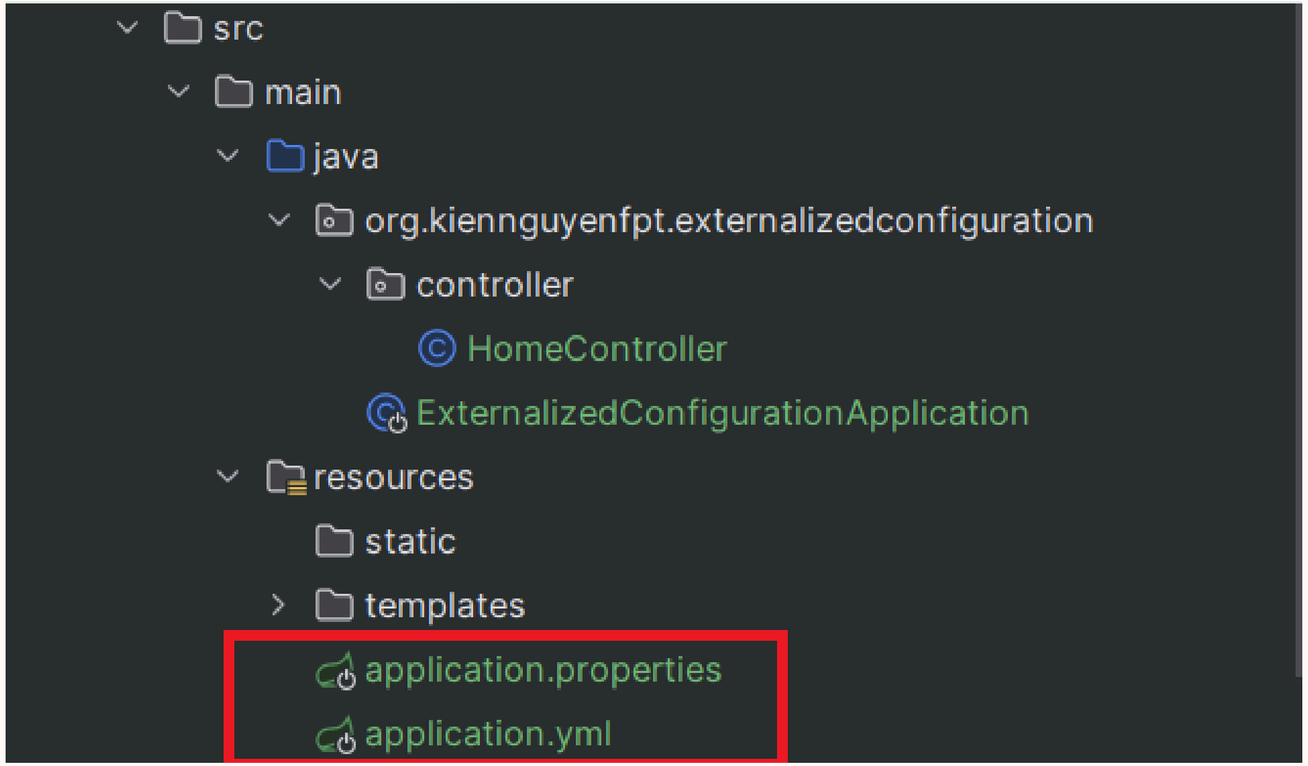
By default, Spring Boot looks for application.properties or application.yml to load configuration properties. If both files are present, application.properties takes precedence over application.properties.

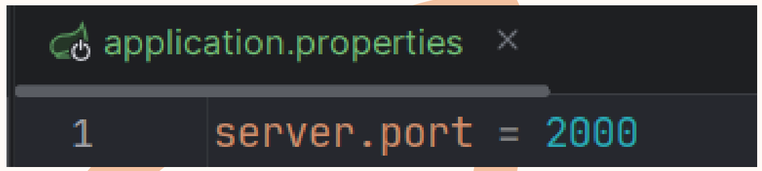


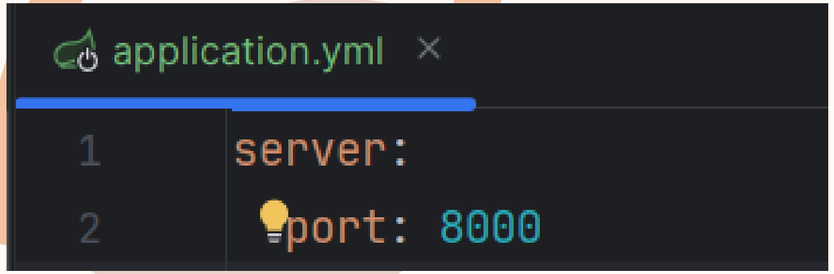


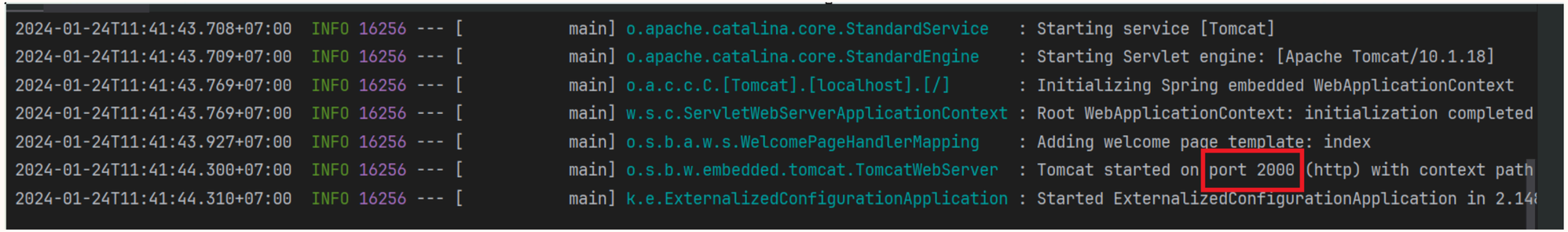


Default Spring Boot Load File:









Some common method to access property value

1) Use @Value annotation

- Load Properties Value: Define property in application.properties or application.yml

application.yml

```yaml
property.name:
  Wellcome to FSOFT!
```

1) Use @Value annotation

- The approach to access property value:

```
@Component
public class MyComponent {
    @Value("${property.name}")
    private String myProp;
    1 usage  new *
    public void printProperty() { System.out.println("My access property: " + myProp); }
}
```

${property.name} is a placeholder that Spring Boot will replace with the value configured in property files.

```
public static void main(String[] args) {
    ApplicationContext context = SpringApplication.run(ExternalizedConfigurationApplication.class, args);
    MyComponent mc = context.getBean(MyComponent.class);
    //access and print the property value
    mc.printProperty();
}
```

1) Use @Value annotation

```java
@Component
public class MyComponent {
    @Value("${property.name}")
    private String myProp;
    1 usage  new *
    public void printProperty() { System.out.println("My access property: " + myProp); }
}
```

```java
public static void main(String[] args) {
    ApplicationContext context = SpringApplication.run(ExternalizedConfigurationApplication.class, args);
    MyComponent mc = context.getBean(MyComponent.class);
    //access and print the property value
    mc.printProperty();
}
```

```
2024-01-24T23:09:07.628+07:00  INFO 20168 --- [
2024-01-24T23:09:07.691+07:00  INFO 20168 --- [
2024-01-24T23:09:07.691+07:00  INFO 20168 --- [
2024-01-24T23:09:07.832+07:00  INFO 20168 --- [
2024-01-24T23:09:08.161+07:00  INFO 20168 --- [
2024-01-24T23:09:08.168+07:00  INFO 20168 --- [
My access property: Wellcome to FSOFT!
```

2) Use Environment object

- Load Properties Value: Define property in application.properties or application.yml

```
application.yml
1  property.name:
2    Wellcome to FSOFT!
```

2) Use Environment

- The approach to access property value:

```
@Component
public class MyComponent {
    @Autowired
    private Environment env;
    1 usage  new *
    public void displayPropValue() {
        String myProp = env.getProperty("property.name");
        System.out.println("My access property value: " + myProp);
    }
}
```

the Environment object is autowired, and the getProperty method is used to retrieve the property value.

2) Use Environment

- The approach to access property value:

```
@Component
public class MyComponent {
    @Autowired
    private Environment env;
    1 usage  new *
    public void displayPropValue() {
        String myProp = env.getProperty("property.name");
        System.out.println("My access property value: " + myProp);
    }
}
```

```
public static void main(String[] args) {
    ApplicationContext context = SpringApplication.run(ExternalizedConfigurationApplication.class, args);
    MyComponent mc = context.getBean(MyComponent.class);
    //access and print the property value
    mc.displayPropValue();
}
```

```java
@Component
public class MyComponent {
    @Autowired
    private Environment env;
    1 usage  new *
    public void displayPropValue() {
        String myProp = env.getProperty("property.name");
        System.out.println("My access property value: " + myProp);
    }
}
```

```java
public static void main(String[] args) {
    ApplicationContext context = SpringApplication.run(ExternalizedConfigurationApplication.class, args);
    MyComponent mc = context.getBean(MyComponent.class);
    //access and print the property value
    mc.displayPropValue();
}
```

```
2024-01-24T23:21:41.753+07:00  INFO 19796 --- [
2024-01-24T23:21:41.807+07:00  INFO 19796 --- [
2024-01-24T23:21:41.807+07:00  INFO 19796 --- [
2024-01-24T23:21:41.953+07:00  INFO 19796 --- [
2024-01-24T23:21:42.329+07:00  INFO 19796 --- [
2024-01-24T23:21:42.338+07:00  INFO 19796 --- [
My access property value: Wellcome to FSOFT!
```

3) Use @ConfigurationProperties

application_job.properties

```
job.application = "intership"
external.prop-name="Trung Kiennnnnnnnnn"
```

3) Use @ConfigurationProperties

```java
@Configuration
@org.springframework.boot.context.properties.ConfigurationProperties(prefix = "external")
@PropertySource({"classpath:application_job.properties", "classpath:application_name.properties"})
public class ConfigurationProperties {
    2 usages
    private String propName;

    new *
    public String getPropName() {
        return propName;
    }

    new *
    public void setPropName(String propName) {
        this.propName = propName;
    }
}
```

@ConfigurationProperties(prefix = "external") specifies that properties in externalized configuration with prefix "external" will be map with field propName

3) Use @ConfigurationProperties

```
public static void main(String[] args) {
    ApplicationContext context = SpringApplication.run(ExternalizedConfigurationApplication.class, args);
    ConfigurationProperties cp = context.getBean(ConfigurationProperties.class);
    System.out.println("Property name from @ConfigurationProperties: " + cp.getPropName());
```

3) Use @ConfigurationProperties

application_job.properties

```
job.application = "intership"
external.prop-name="Trung Kiennnnnnnnnn"
```

The ouput:

```
2024-01-25T12:17:02.720+07:00  INFO 25464 --- [          main] o.ap
2024-01-25T12:17:02.770+07:00  INFO 25464 --- [          main] o.a.
2024-01-25T12:17:02.770+07:00  INFO 25464 --- [          main] w.s.
2024-01-25T12:17:02.921+07:00  INFO 25464 --- [          main] o.s.
2024-01-25T12:17:03.214+07:00  INFO 25464 --- [          main] o.s.
2024-01-25T12:17:03.223+07:00  INFO 25464 --- [          main] k.e.
Property name from @ConfigurationProperties: "Trung Kiennnnnnnnnn"
```

The approach to load properties from multiple files

1) Create multiple configuration files

Suppose that we have 2 .properties files: job_application.properties and name_application.properties and we want to load all these below properties:

application_job.properties

```
job.application = "intership"
```

application_name.properties

```
name.application = "Trung Kien"
```

2) Use @PropertySource in Configuration Class

```
@Configuration
@PropertySource({"classpath:application_job.properties", "classpath:application_name.properties"})
public class ConfigurationProperties {
}
```

@PropertySource is used to specify an array of file locations

3) Access Properties

```java
@Component
public class MyComponent {
    @Value("${job.application}")
    private String job;

    @Value("${name.application}")
    private String name;

    1 usage  new *
    public void displayMultipleProperties() {
        System.out.println("Job property: " + job);
        System.out.println("Name property: " + name);
    }
```

when we runapplication, it will load properties from both application-defaults.properties and application-custom.properties. The @Value annotation is used to inject the values of properties into the MyComponent bean.

Here is the output:

```
2024-01-25T11:35:06.606+07:00  INFO 26976 ---
2024-01-25T11:35:06.606+07:00  INFO 26976 ---
2024-01-25T11:35:06.724+07:00  INFO 26976 ---
2024-01-25T11:35:07.057+07:00  INFO 26976 ---
2024-01-25T11:35:07.067+07:00  INFO 26976 ---
Job property: "intership"
Name property: "Trung Kien"
```

Fetch values from system property or environment variable

1) Use @Value annotation

```
@Component
public class MyComponent {
    @Value("${external.api.key:DefaultValueee}")
    private String apiKey;
    1 usage  new *
    public void displayAPIKey() {
        System.out.println("Value of API: " + apiKey);
    }
```

${external.api.key:defaultValueee} is a placeholder that tries to fetch the value of the system property named external.api.key
If the system property is not set, it defaults to "defaultValueee".

2) Use Environment

```
@Component
public class MyComponent {
    1 usage
    private Environment env;
    no usages  new *
    public void displayProp() {
        String propVal =  env.getProperty( key: "external_api_key",  defaultValue: "defaultValue");
        System.out.println("Value of API key: " + propVal);
    }
```

We use Environment object, and the getProperty method is used to fetch the value of the system property named “external_api_key”. If the system property is not set, it defaults to "defaultValue".

3) Use Main method to fetch environment variable

Suppose we have environment variable name “my_key” and we can use main method to fetch the value of “my_key”

```
public static void main(String[] args) {
    SpringApplication.run(ExternalizedConfigurationApplication.class, args);
    System.out.println(System.getenv( name: "my_key"));
```

How to provide default value for the placeholder

The simple way is to use @Value

- Syntax:

```
@Value("{propertyName:defaultValue}")
private String propertyName;
```

- For example:

```
@Value("${external.api.key:DefaultValueee}")
private String apiKey;
1 usage  new *
public void displayAPIKey() {
    System.out.println("Value of API: " + apiKey);
}
```

The simple way is to use @Value

```
@Value("${external.api.key:DefaultValueee}")
private String apiKey;
1 usage  new *
public void displayAPIKey() {
    System.out.println("Value of API: " + apiKey);
}
```

In this case, if external.api.key is not configured in configuration file, the apiKey will default to "DefaultValueee"

2) Use directly on file .yml or .properties

```
property.name:
  Wellcome to FSOFT!
```

Some approaches to handle missing property file

Handling missing property file

- Handle missing property file in Spring Boot implies dealing with situations where a configuration or property file is not present in a Spring Boot application.

- This work typically involves setting default values or providing alternative configurations to ensure that the application can run smoothly even if the expected properties are not present.

Scenario

Suppose I am developing a Spring Boot application that requires an API key to communicate with an external service. This API key is configured in application.properties file. However, in a development environment, I want to run the application without an API key, and if an API key is not configured, the application will automatically switch to a "no API key" environment.

1) Use @Value annotation

```yaml
external:
api:
key:
```

```java
@Component
public class MyComponent {
    @Value("${external.api.key:DefaultValueee}")
    private String apiKey;
    1 usage  new *
    public void displayAPIKey() { System.out.println("Value of API: " + apiKey); }

}
```

The output is:

```
2024-01-25T10:24:37.637+07:00  INFO 25088 --- [
2024-01-25T10:24:37.669+07:00  INFO 25088 --- [
2024-01-25T10:24:37.670+07:00  INFO 25088 --- [
2024-01-25T10:24:37.743+07:00  INFO 25088 --- [
2024-01-25T10:24:37.880+07:00  INFO 25088 --- [
2024-01-25T10:24:37.885+07:00  INFO 25088 --- [
Value of API: DefaultValueee
```

2) Use @ConditionalProperties

application.yml

```yaml
external:
api:
key:
```

```java
@Component
@org.springframework.boot.context.properties.ConfigurationProperties(prefix = "external.api")
public class ConfigurationProperties {
    2 usages
    private String apiKey = "fsoft.com";

    new *
    public String getApiKey() {
        return apiKey;
    }

    new *
    public void setApiKey(String apiKey) {
        this.apiKey = apiKey;
    }
}
```

2) Use @ConditionalProperties

```
@Component
public class MyComponent {
    2 usages
    ConfigurationProperties cp = new ConfigurationProperties();
    new *
    @Autowired
    public MyComponent(ConfigurationProperties cp) {
        this.cp = cp;
    }
    1 usage  new *
    public void displayAPIKey() {
        System.out.println("Value of API: " + cp.getApiKey());
    }
}
```

2) Use @ConditionalProperties

```java
@Component
@org.springframework.boot.context.properties.ConfigurationProperties(prefix = "external.api")
public class ConfigurationProperties {
    2 usages
    private String apiKey = "fsoft.com";

    new *
    public String getApiKey() {
        return apiKey;
    }

    new *
    public void setApiKey(String apiKey) {
        this.apiKey = apiKey;
    }
}
```

```java
@Component
public class MyComponent {
    2 usages
    ConfigurationProperties cp = new ConfigurationProperties();
    new *
    @Autowired
    public MyComponent(ConfigurationProperties cp) {
        this.cp = cp;
    }
    1 usage  new *
    public void displayAPIKey() {
        System.out.println("Value of API: " + cp.getApiKey());
    }
}
```

```
2024-01-25T10:37:56.549+07:00  INFO 26996 --- [
2024-01-25T10:37:56.588+07:00  INFO 26996 --- [
2024-01-25T10:37:56.589+07:00  INFO 26996 --- [
2024-01-25T10:37:56.646+07:00  INFO 26996 --- [
2024-01-25T10:37:56.788+07:00  INFO 26996 --- [
2024-01-25T10:37:56.793+07:00  INFO 26996 --- [
Value of API: fsoft.com
```

3) Checking for Property Existence:

```
@Component
public class MyComponent {
    @Value("${external.api.key}")
    private String apiKey;
    1 usage  new *
    public void displayAPIKey() {
        if(apiKey.isEmpty()) {
            System.out.println("No API key found!");
        } else System.out.println("Value of API: " + apiKey);
    }
}
```

3) Checking for Property Existence:

```java
@Component
public class MyComponent {
    @Value("${external.api.key}")
    private String apiKey;
    1 usage  new *
    public void displayAPIKey() {
        if(apiKey.isEmpty()) {
            System.out.println("No API key found!");
        } else System.out.println("Value of API: " + apiKey);
    }
}
```

```
2024-01-25T10:53:26.418+07:00  INFO 15804 --- [
2024-01-25T10:53:26.477+07:00  INFO 15804 --- [
2024-01-25T10:53:26.617+07:00  INFO 15804 --- [
2024-01-25T10:53:26.622+07:00  INFO 15804 --- [
No API key found!
```

Special thank you for your listening!