SME Walkthrough

1. Project Structure

When you extract the project, you'll see a standard Maven project layout:

- `src/main/java`:
    - **Purpose:** This folder contains your primary Java source code for the application. All your domain classes, service classes, controllers, and the main application entry point (`SpringLearnApplication.java`) reside here.
    - **Example:** `com/cognizant/springlearn/SpringLearnApplication.java`
- `src/main/resources`:
    - **Purpose:** This folder is for application configuration, static assets, templates, and other non-Java resources. Spring Boot automatically picks up files from this location.
    - **Common Files:**
        - `application.properties` or `application.yml`: For externalized configuration (e.g., database connection details, server port).
        - `static/`: For static web content (HTML, CSS, JavaScript, images).
        - `templates/`: For server-side templates (e.g., Thymeleaf, Freemarker).
    - **Example:** You'll find an empty `application.properties` file here by default.
- `src/test/java`:
    - **Purpose:** This folder contains Java source code for unit and integration tests of your application. It mirrors the package structure of `src/main/java`.
    - **Example:** `com/cognizant/springlearn/SpringLearnApplicationTests.java` (a basic test class generated by Spring Initializr).
- `pom.xml`:
    - **Purpose:** The Project Object Model (POM) file is the core configuration file for Maven projects. It defines project dependencies, build plugins, project metadata, and more.
- `.mvn`:
    - **Purpose:** Contains Maven wrapper scripts (`mvnw`, `mvnw.cmd`) and the `wrapper` directory with the Maven wrapper JAR. This allows you to run Maven commands without having Maven explicitly installed on your system, using the bundled version.
- `HELP.md`:
    - **Purpose:** A markdown file providing basic instructions and links for the Spring Boot project.

2. `SpringLearnApplication.java` - `main()` Method

Java

```java
package com.cognizant.springlearn;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringLearnApplication {

    private static final Logger LOGGER =
LoggerFactory.getLogger(SpringLearnApplication.class);

    public static void main(String[] args) {
        SpringApplication.run(SpringLearnApplication.class, args);
        LOGGER.info("SpringLearnApplication started successfully!");
    }
}
```

- **`public static void main(String[] args)`**: This is the standard entry point for any Java application. When you run your Spring Boot application, this method is executed first.
- **`SpringApplication.run(SpringLearnApplication.class, args);`**: This is the magical line that bootstraps your Spring Boot application.
  - It creates an `ApplicationContext`, which is the core of the Spring IoC (Inversion of Control) container.
  - It performs component scanning to find and register all Spring components (like `@Controller`, `@Service`, `@Repository`, `@Component`).
  - It configures embedded web servers (like Tomcat, which comes with `spring-boot-starter-web`) if `spring-web` is on the classpath.
  - It manages the entire lifecycle of your Spring application.
- **`LOGGER.info(...)`**: As added, this line simply prints a message to the console once the Spring application has finished its startup sequence. This confirms that your application has successfully initialized.

3. Purpose of `@SpringBootApplication` Annotation

The `@SpringBootApplication` annotation is a convenience annotation that combines three commonly used Spring Boot annotations:

- **`@SpringBootConfiguration`**: Tags the class as a source of bean definitions for the application context. This is essentially equivalent to Spring's `@Configuration` annotation, indicating that the class can define `@Bean` methods.
- **`@EnableAutoConfiguration`**: This is the key annotation that enables Spring Boot's auto-configuration mechanism. Based on the JAR dependencies you've included in your project (e.g., `spring-boot-starter-web`), Spring Boot intelligently guesses and configures what you need. For instance, if you have `spring-web`, it auto-configures Tomcat, DispatcherServlet, etc.
- **`@ComponentScan`**: This tells Spring to scan for components (classes annotated with `@Component`, `@Service`, `@Repository`, `@Controller`, etc.) within the package where `@SpringBootApplication` is located (and its sub-packages). This allows Spring to automatically discover and register your application's components as beans.

In essence, `@SpringBootApplication` makes it easy to set up a basic Spring Boot application with minimal configuration.

4. `pom.xml` - Configuration Walkthrough

The `pom.xml` file is crucial for managing your project. Let's look at the key sections:

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.3.1</version> <relativePath/> </parent>

    <groupId>com.cognizant</groupId>
    <artifactId>spring-learn</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>spring-learn</name>
    <description>Demo project for Spring Boot</description>

    <properties>
```

```
        <java.version>17</java.version> </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <scope>runtime</scope>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

- **`<project>`** : The root element of the POM.
- **`<modelVersion>`** : Specifies the version of the Maven model being used (always 4.0.0 for current Maven versions).
- **`<parent>`** :
  - This is crucial for Spring Boot projects. It inherits configurations from `spring-boot-starter-parent` .
  - **`groupId`** , **`artifactId`** , **`version`** : Identify the parent POM.
  - **`relativePath`** : Indicates where to find the parent POM if it's not in a local repository (usually left empty to let Maven find it).

- **Benefit:** This parent POM provides default configurations for Java version, encoding, resource filtering, and critically, **dependency management**. This means you don't need to specify versions for most Spring Boot dependencies (they are managed by the parent).

- `<groupId>`, `<artifactId>`, `<version>`: These three elements together form the unique coordinates of your project (GAV coordinates).
    - `groupId`: The unique identifier of the organization or group (e.g., `com.cognizant`).
    - `artifactId`: The unique identifier of the project within the group (e.g., `spring-learn`).
    - `version`: The current version of your project. `0.0.1-SNAPSHOT` indicates a "snapshot" or development version.
- `<name>`: A more readable name for your project.
- `<description>`: A brief description of the project.
- `<properties>`:
    - Defines various properties that can be reused throughout the POM.
    - `<java.version>`: Specifies the Java compatibility version for your project. Maven will use this to compile your source code.
- `<dependencies>`:
    - This section declares all the external libraries (dependencies) that your project needs to compile and run.
    - `<dependency>`: Each dependency block defines a single library.
        - `spring-boot-starter-web`: This is a "starter" dependency for building web applications (including RESTful services) using Spring MVC. It brings in embedded Tomcat, Spring MVC, Jackson (for JSON), and other web-related libraries.
        - `spring-boot-devtools`: Provides development-time features like automatic application restarts when code changes, LiveReload support, and more.
            - `<scope>runtime</scope>`: Means this dependency is only needed at runtime, not for compilation.
            - `<optional>true</optional>`: Prevents this dependency from being transitively included by other projects that depend on `spring-learn`.
        - `spring-boot-starter-test`: A starter for testing Spring Boot applications. It includes popular testing libraries like JUnit, Mockito, AssertJ, and Spring Test.
            - `<scope>test</scope>`: Means this dependency is only available for compiling and running tests, not for the main application's runtime.
- `<build>`:
    - This section configures the build process of your project.
    - `<plugins>`: Defines the Maven plugins used during the build.

- `spring-boot-maven-plugin` : This is the most important plugin for Spring Boot applications.
    - It collects all the JARs on the classpath and builds a single, executable "fat JAR" or "uber JAR" that contains all dependencies.
    - It provides goals to run your application (`mvn spring-boot:run`), repackage (`mvn package`), and build info.

5. Open 'Dependency Hierarchy' in IntelliJ IDEA

The Dependency Hierarchy view in IntelliJ IDEA is incredibly useful for understanding the transitive dependencies of your project and identifying potential dependency conflicts.

1. In IntelliJ IDEA, open your `pom.xml` file.
2. Look for a tool window tab at the bottom or right side of the IDE labeled **"Maven"** (or sometimes just "Maven Projects"). Click on it to open the Maven tool window.
3. In the Maven tool window, expand your project (`spring-learn`).
4. Under your project, you'll see a section like "Dependencies".
5. Click on the **"Dependency Analyzer"** (often represented by a magnifying glass or graph icon).
    - Alternatively, sometimes there's a "Show Dependencies" or "Dependency Hierarchy" view directly available.
6. The "Dependency Analyzer" will display a graphical or tree-like representation of all your project's dependencies, including transitive ones (dependencies of your dependencies).
    - You can search for specific dependencies.
    - You can filter by scope (compile, test, runtime).
    - You can see which dependencies are conflicting or excluded.

This view helps you understand why certain libraries are included in your build and can be invaluable for debugging classpath issues.