Spring Boot Java framework: cheat sheet of maven pom.xml

Data Backend Tech

.

Follow

5 min read

.

May 28, 2023

--

1

Listen

Share

Introduction

The `pom.xml` file is a crucial configuration file in a Maven-based Spring Boot project. It defines project-specific information, dependencies, plugins, and other build-related configurations. Understanding and effectively utilizing the `pom.xml` file is essential for managing dependencies, controlling the build process, and ensuring project stability. This comprehensive cheat sheet provides a handy reference guide for various elements and configurations that can be used in the `pom.xml` file of a Spring Boot project.

Project Information

Define the project's group ID, artifact ID, and version:

```
<groupId>com.example</groupId>
<artifactId>my-project</artifactId>
<version>1.0.0</version>
```

Dependencies

Add dependencies required for your project:

```
<dependencies>
 <dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
   <version>2.5.0</version>
 </dependency>
 <!-- Add more dependencies as needed -->
</dependencies>
```

In Maven's `pom.xml` file, the `<scope>` element is used to define the scope of a dependency. The scope determines how the dependency is used during different phases of the build process and runtime. Maven provides different dependency scopes to control the visibility and accessibility of dependencies. Here are some commonly used dependency scopes in Maven:

1. `compile` (default scope):

- Dependencies with this scope are required for compiling and running the application.
- They are available on the classpath during the compile, test, and runtime phases.
- Example: `<scope>compile</scope>`

2. `provided`:

- Dependencies with this scope are required for compiling and running the application, but they are expected to be provided by the runtime environment.
- They are available on the classpath during the compile and test phases but are not packaged with the application.
- Example: `<scope>provided</scope>`

3.`runtime`:

- Dependencies with this scope are required for running the application but not for compiling.
- They are not available on the classpath during the compile phase but are available during the test and runtime phases.
- Example: `<scope>runtime</scope>`

4.`test`:

- Dependencies with this scope are only used for testing purposes.
- They are available on the classpath during the test phase but are not included in the runtime classpath or packaged with the application.
- Example: `<scope>test</scope>`

5. `system`:

- Dependencies with this scope are similar to `compile` dependencies but are referenced using an explicit path on the local system.
- They are not available in any Maven repository.
- Example:

```
<scope>system</scope>
<systemPath>/path/to/dependency.jar</systemPath>
```

6.`import`:

- Dependencies with this scope are used only for importing other Maven projects.
- They are not used for compilation, testing, or runtime.
- Example: `<scope>import</scope>`

By utilizing these dependency scopes in Maven's `pom.xml`, wecan effectively manage the inclusion and visibility of dependencies, optimize the build process, and ensure the correct dependencies are available during different phases of the project lifecycle.

Plugins

Configure plugins for various purposes, such as building, testing, and packaging:

```
<build>
 <plugins>
   <plugin>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-maven-plugin</artifactId>
     <version>2.5.0</version>
   </plugin>
   <!-- Add more plugins as needed -->
 </plugins>
</build>
```

In Maven, plugins are essential components that extend the build process and provide additional functionality. Plugins in Maven are defined within the `<plugins>` section of the `pom.xml` file and can perform a variety of tasks such as compiling code, running tests, generating documentation, deploying artifacts, and more. Here are some key aspects of using plugins in Maven:

Plugin Configuration

- Each plugin has its own configuration parameters that define how it should execute.
- The configuration is specified within the `<configuration>` element of the plugin.
- Example:

```
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-compiler-plugin</artifactId>
 <version>3.8.1</version>
 <configuration>
   <source>1.8</source>
   <target>1.8</target>
 </configuration>
</plugin>
```

Plugin Execution

- Plugins can be bound to specific phases of the build lifecycle using the `<executions>` element.
- Each execution can define one or more goals to be executed in a specific order.
- Example:

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-surefire-plugin</artifactId>
   <version>3.0.0-M5</version>
   <executions>
       <execution>
           <id>run-tests</id>
           <goals>
               <goal>test</goal>
           </goals>
       </execution>
    </executions>
</plugin>
```

Plugin Dependency

- Plugins can have their own dependencies, which are specified within the `<dependencies>` element.
- These dependencies are separate from project dependencies and are used to support the plugin's functionality.
- Example:

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-compiler-plugin</artifactId>
   <version>3.8.1</version>
   <dependencies>
       <dependency>
         <groupId>com.example</groupId>
         <artifactId>custom-compiler</artifactId>
         <version>1.0.0</version>
       </dependency>
   </dependencies>
</plugin>
```

Plugin Management

- Plugin management allows defining plugin versions and configurations in a central location within the `<pluginManagement>` section.
- This allows for consistent configuration across multiple projects.
- Example:

```
<pluginManagement>
   <plugins>
       <plugin>
           <groupId>org.apache.maven.plugins</groupId>
           <artifactId>maven-compiler-plugin</artifactId>
           <version>3.8.1</version>
           <configuration>
               <source>1.8</source>
               <target>1.8</target>
           </configuration>
       </plugin>
   </plugins>
</pluginManagement>
```

Plugins are a powerful feature of Maven that enable developers to extend the build process and customize various aspects of the project. They provide a wide range of functionality and can be easily integrated into a Maven project by specifying the appropriate plugin coordinates and configuration in the `pom.xml` file.

For more details of use cases for plugin, please also refer to another article:
https://medium.com/@databackendtech/spring-boot-java-framework-when-to-use-maven-plugin-95a9cfd6257c

Profiles

Define profiles for different environments or build configurations:

```
<profiles>
 <profile>
   <id>dev</id>
   <properties>
     <env>dev</env>
```

```
   </properties>
 </profile>
 <!-- Add more profiles as needed -->
</profiles>
```

Resource Filtering

Enable resource filtering to replace placeholders in resource files:

```
<build>
 <resources>
   <resource>
     <directory>src/main/resources</directory>
     <filtering>true</filtering>
   </resource>
 </resources>
</build>
```

Custom Properties

Define custom properties for reuse and easy management:

```
<properties>
 <java.version>11</java.version>
 <spring.boot.version>2.5.0</spring.boot.version>
</properties>
```

Parent pom.xml

We can also use parent `pom.xml` to define common configurations and dependencies shared across multiple child projects. It allows for centralized management of project settings and promotes code reuse and consistency.

When to Use Parent `pom.xml`

We may consider using a parent `pom.xml` in the following scenarios:

1. **Shared Configurations**: If there are multiple projects that require the same configuration settings, such as the Java version, encoding, or resource filtering, we can define these settings in the parent `pom.xml` to avoid duplication.
2. **Shared Dependencies**: If there is a set of dependencies that are common across multiple projects, we can define them in the parent `pom.xml`. This ensures that all child projects inherit the same versions of the dependencies, promoting consistency.
3. **Shared Plugins**: If we use plugins in the build process, such as the Spring Boot Maven Plugin or code generation plugins, we can define them in the parent `pom.xml`. This allows all child projects to use the same versions and configurations of these plugins.

Where to Put the Parent `pom.xml`

The parent `pom.xml` can be placed in a separate directory or repository, distinct from the child projects. It is common to create a dedicated folder or repository to host the parent `pom.xml`. This allows for easy management and versioning of the parent project.

How to Use the Parent `pom.xml` in Child Projects

To use the parent `pom.xml` in the child projects, we can follow these steps:

1. Create a new `pom.xml` file in the child project if it doesn't exist already.
2. In the child project's `pom.xml`, add the `<parent>` element under the root `<project>` element:

```
<parent>
 <groupId>com.example</groupId>
 <artifactId>my-parent-project</artifactId>
 <version>1.0.0</version>
</parent>
```

- Replace the values with the appropriate coordinates of your parent project.
- If the parent `pom.xml` is located in a different directory or repository, you need to specify the relative path to the parent `pom.xml` using the `<relativePath>` element:

```
<parent>
 <groupId>com.example</groupId>
```

```
 <artifactId>my-parent-project</artifactId>
 <version>1.0.0</version>
 <relativePath>../my-parent-project/pom.xml</relativePath>
</parent>
```

- Adjust the relative path according to the real project’s directory structure.

3. Customize the child project’s `pom.xml` as needed. Any configurations or dependencies defined in the child project's `pom.xml` will override those inherited from the parent `pom.xml`.

Summary

The `pom.xml` file in a Spring Boot project plays a vital role in managing dependencies, configuring plugins, and controlling the build process. This comprehensive cheat sheet provides a quick reference guide for the essential elements and configurations that can be included in the `pom.xml` file. By utilizing this cheat sheet, developers can effectively manage their project's dependencies, plugins, profiles, and resource filtering, ultimately ensuring a smooth and efficient build process in their Spring Boot applications.