DevOps Lab - Trisha Balakrishnan

Exercise 6

Question 1: Customizing Maven Build Lifecycle

Your team requires that unit tests should only be executed during the integration-test phase and not during the standard test phase, which is part of the default Maven lifecycle.

How would you modify the Maven build lifecycle to achieve this requirement? What changes would you make in the pom.xml file? To execute unit tests during the `integration-test` phase rather than the `test` phase, modify the `maven-surefire-plugin` configuration in the pom.xml to skip tests during the test phase and bind the `maven-surefire-plugin` to the `integration-test` phase:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>3.0.0-M5</version>
      <configuration>
        <skipTests>true</skipTests> <!-- Skips tests in the test phase -->
      </configuration>
      <executions>
        <execution>
          <id>run-tests-in-integration-test-phase</id>
          <phase>integration-test</phase> <!-- Runs tests in the integration-test phase -->
          <goals>
            <goal>test</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Question 2: Adding External Dependencies and Managing Versions

Your project requires the latest stable version of the Spring Framework, but the repository's current version is outdated. You also need to ensure that the same version of the Spring Framework is used across all modules of your multi-module project.

How would you manage the external dependency in your pom.xml and ensure version consistency across multiple modules? To execute unit tests during the integration-test phase rather than the test phase, modify the maven-surefire-plugin configuration in the pom.xml to skip tests during the test phase and bind the maven-surefire-plugin to the integration-test phase: To ensure the latest stable version of the Spring Framework is used across all modules, use a `dependencyManagement` section in the parent `pom.xml` to define the version:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>${spring.version}</version>
    </dependency>
  </dependencies>
</dependencyManagement>

<properties>
  <spring.version>5.3.21</spring.version> <!-- Define the Spring version once -->
</properties>
```

Question 3: Skipping Tests During Build

You are in a rush to deploy a hotfix to production. Running the full suite of unit and integration tests is time-consuming, and you want to skip them during the build process for this hotfix.

How would you configure Maven to skip tests during the build? Is there a specific command you can use to achieve this? To skip tests during the build for a hotfix, either add the following configuration in the pom.xml:

```
<properties>
  <maven.test.skip>true</maven.test.skip>
</properties>
```

Alternatively, you can skip tests by using the following Maven command:

```
mvn clean install -DskipTests
```

Question 4: Managing Plugins for Code Coverage

Your project team wants to ensure that the code coverage is above 80% and wants to integrate code coverage reports as part of the build process.

How would you integrate a code coverage tool like JaCoCo into the Maven build lifecycle, and ensure it breaks the build if the coverage falls below 80%? To integrate JaCoCo and fail the build if the coverage is below 80%, add the JaCoCo plugin to your pom.xml:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.jacoco</groupId>
      <artifactId>jacoco-maven-plugin</artifactId>
      <version>0.8.7</version>
      <executions>
        <execution>
          <goals>
            <goal>prepare-agent</goal>
          </goals>
        </execution>
        <execution>
          <id>report</id>
          <phase>verify</phase>
          <goals>
            <goal>report</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <groupId>org.jacoco</groupId>
      <artifactId>jacoco-check-maven-plugin</artifactId>
      <version>0.8.7</version>
      <executions>
        <execution>
          <goals>
            <goal>check</goal>
          </goals>
          <configuration>
            <rules>
              <rule>
                <element>BUNDLE</element>
                <limits>
                  <limit>
                    <counter>LINE</counter>
                    <value>COVEREDRATIO</value>
                    <minimum>0.80</minimum> <!-- Fail if coverage is below 80% -->
                  </limit>
                </limits>
              </rule>
            </rules>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Question 5: Managing Multi-Module Project Dependencies

You are working on a multi-module project where Module A is dependent on Module B. You need to ensure that any changes in Module B are automatically reflected in Module A without requiring external builds or manual version management.

How would you structure the pom.xml files for both modules to handle this dependency? How would you ensure that changes in Module B are integrated into Module A during the build?
To manage dependencies between Module A and Module B, set up Module A's pom.xml to depend on Module B like this:

```
<dependencies>
  <dependency>
    <groupId>com.example</groupId>
    <artifactId>module-b</artifactId>
    <version>${project.version}</version> <!-- Use the same version as Module B -->
  </dependency>
</dependencies>
```

Both modules should inherit from a common parent pom.xml to ensure that changes in Module B automatically propagate to Module A during the build.

Question 6: Overriding a Dependency Version in a Child Module

You are using a third-party library (e.g., Hibernate) in both parent and child modules of your project. However, the child module needs to use a different version of Hibernate from the one declared in the parent POM.

How would you override the dependency version in the child module's pom.xml without affecting the parent module? In the child module's pom.xml, override the dependency version like this:

```
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.4.32.Final</version> <!-- Override version in child module -->
  </dependency>
</dependencies>
```

This will only affect the child module, leaving the parent unchanged.

Scenario 7: Excluding Transitive Dependencies

You have a project that includes a third-party library, but the library comes with unwanted transitive dependencies that are causing conflicts with your existing dependencies. You want to exclude these transitive dependencies.

How would you exclude a specific transitive dependency from a third-party library in your pom.xml? To exclude specific transitive dependencies in a Maven project, you can use the `<exclusions>` tag within the `<dependency>` tag of your pom.xml. Here's an example of how to exclude a transitive dependency:

```
<dependency>
    <groupId>com.example</groupId>
    <artifactId>example-library</artifactId>
    <version>1.0.0</version>
    <exclusions>
        <exclusion>
            <groupId>com.unwanted</groupId>
            <artifactId>unwanted-library</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

In this example, the `unwanted-library` from `com.unwanted` is excluded from being included as a transitive dependency of `example-library`. ### Question 7: Using Maven Properties to Manage Project Information #### You want to reuse specific information, such as the project version or database URL, across multiple places in the pom.xml. Instead of hardcoding values, you want to manage them using Maven properties. #### How would you define and use Maven properties in the pom.xml to avoid duplicating values across different configurations? (Use properties tag) To manage project information like version or database URLs using Maven properties, you can define properties in the `<properties>` section of the pom.xml. You can then reuse these properties throughout the POM by referencing them with `${property-name}`.

Here's an example:

```
<properties>
    <project.version>1.0.0</project.version>
    <db.url>jdbc:mysql://localhost:3306/mydb</db.url>
</properties>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>${project.version}</version>
```

```xml
</dependency>

<configuration>
    <url>${db.url}</url>
</configuration>
```

This setup allows you to manage the version and database URL in one place and reuse it across the POM.

Question 8: Enforcing a Minimum Java Version for the Project

Your project requires a minimum version of Java (e.g., Java 11) to build successfully. You want Maven to fail the build if a lower version of Java is used.

How would you configure the pom.xml to ensure that the project only builds with Java 11 or higher? To ensure Maven builds the project only with Java 11 or higher, you can use the `maven-enforcer-plugin` to specify a minimum Java version. Here's how to configure it:

```xml
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-enforcer-plugin</artifactId>
            <version>3.0.0</version>
            <executions>
                <execution>
                    <goals>
                        <goal>enforce</goal>
                    </goals>
                    <configuration>
                        <rules>
                            <requireJavaVersion>
                                <version>[11,)</version>
                            </requireJavaVersion>
                        </rules>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

This will fail the build if the Java version is lower than 11.

Question 9: Customizing the Default Build Lifecycle

You are working on a Maven project where your team has decided that static code analysis using Checkstyle should be enforced before the unit tests are executed. Currently, Checkstyle is not part of the default Maven lifecycle.

How would you modify the build lifecycle to ensure that Checkstyle runs before the test phase? Describe the changes needed in the pom.xml and explain how the Maven build lifecycle would work after the changes. To run Checkstyle before the test phase, you can configure the `maven-checkstyle-plugin` and bind it to the `validate` phase (which occurs before the test phase). Here's how to do it:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-checkstyle-plugin</artifactId>
            <version>3.1.1</version>
            <executions>
                <execution>
                    <id>checkstyle-validation</id>
                    <phase>validate</phase>
                    <goals>
                        <goal>check</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

This ensures that Checkstyle runs during the `validate` phase, which is before the `test` phase in the Maven lifecycle.

Question 10: Skipping Specific Lifecycle Phases

You are preparing a quick build for a production hotfix. Your team has decided to skip running unit tests temporarily, but you still want to perform integration tests.

How would you skip the test phase and still execute the integration-test and verify phases in your Maven build? What commands and configurations would you use? To skip the test phase but still execute the integration-test and verify phases, you can use the following Maven command:

```
mvn verify -DskipTests
```

This will skip the unit tests (test phase) but still run the `integration-test` and `verify` phases. You can configure this in your POM by setting the `maven-surefire-plugin` to skip tests:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <configuration>
                <skipTests>true</skipTests>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Question 11: Binding Goals to Custom Phases

In your project, you need to perform a database schema migration using Flyway right before the integration-test phase. The migration should only be performed in certain environments (e.g., staging or production).

How would you configure the Maven lifecycle and bind the Flyway plugin to run during a custom phase, ensuring it only runs for the correct environments? To bind the Flyway migration to a specific phase (like just before the `integration-test` phase) and ensure it only runs in certain environments (e.g., staging or production), you can configure profiles and bind the Flyway plugin like this:

```xml
<profiles>
    <profile>
        <id>staging</id>
        <build>
            <plugins>
                <plugin>
                    <groupId>org.flywaydb</groupId>
                    <artifactId>flyway-maven-plugin</artifactId>
                    <version>7.0.0</version>
                    <executions>
                        <execution>
                            <phase>pre-integration-test</phase>
                            <goals>
                                <goal>migrate</goal>
                            </goals>
                        </execution>
                    </executions>
                </plugin>
            </plugins>
        </build>
    </profile>
    <profile>
        <id>production</id>
        <build>
            <plugins>
                <plugin>
                    <groupId>org.flywaydb</groupId>
                    <artifactId>flyway-maven-plugin</artifactId>
                    <version>7.0.0</version>
                    <executions>
                        <execution>
                            <phase>pre-integration-test</phase>
                            <goals>
                                <goal>migrate</goal>
                            </goals>
                        </execution>
                    </executions>
                </plugin>
            </plugins>
        </build>
    </profile>
</profiles>
```

Activate the profile when building for the respective environment:

```
mvn clean verify -Pstaging
```

Question 12: Handling Failed Builds

During the build process, you notice that your Maven project fails during the package phase due to a compilation error in one of the modules. However, the other modules compile and package successfully.

How would you troubleshoot and isolate the cause of the build failure? Which Maven commands would you use to focus on the failing module without rerunning the entire build?

(Key : Usage of -pl (projects list) and -am (also make) options to run the build for the affected module and its dependencies only) To isolate a build failure in a specific module, you can use the `-pl` (project list) and `-am` (also make) options to build only the failing module and its dependencies:

```
mvn clean package -pl my-failing-module -am
```

This command ensures that only the my-failing-module and its dependencies are rebuilt without triggering the entire project.

Question 13: Using the Maven Site Lifecycleies and Managing Versions

Your team wants to generate a project site with documentation, code reports, and dependency analysis. You need to ensure that the site is generated automatically after the build is successful.

How would you configure Maven to automatically generate a project site as part of the build process? What goals or phases would you bind the site generation to? To automatically generate a project site after a successful build, you can bind the Maven Site plugin to the `install` or `deploy` phase. Here's how to configure it:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-site-plugin</artifactId>
            <version>3.9.1</version>
            <executions>
                <execution>
                    <id>default-site</id>
                    <phase>install</phase>
                    <goals>
                        <goal>site</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

In this configuration, Maven will generate the site after the `install` phase. You can also bind it to the `deploy` phase for more complex workflows. ### Question 14: Multi-Module Build with Independent Modules #### You have a multi-module Maven project, but some modules are independent and do not rely on others. You want to be able to build certain modules independently without building the entire project. #### How would you structure your pom.xml and use Maven commands to allow independent builds of specific modules? In a multi-module project, to build specific modules independently, you can structure the parent pom.xml as follows:

```
<modules>
    <module>module-a</module>
    <module>module-b</module>
</modules>
```

You can then use the `-pl` (project list) option to build individual modules:

```
mvn install -pl module-a
```

This command will build only `module-a` without building the entire project.

Question 15: Resolving Dependency Conflicts

Your project is facing issues due to conflicting transitive dependencies. Multiple versions of the same library are being pulled in by different dependencies, causing build failures or unexpected runtime behavior.

How would you resolve this dependency conflict in Maven? What are the possible strategies for managing different versions of the same library in the pom.xml? To resolve conflicting dependencies, you can use one of the following strategies: 1. Use in the parent POM to define a single version of the library:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.example</groupId>
            <artifactId>conflicting-library</artifactId>
            <version>1.0.0</version>
        </dependency>
    </dependencies>
</dependencyManagement>
```

2. Exclude the conflicting transitive dependency from specific dependencies:

```
<dependency>
    <groupId>com.example</groupId>
    <artifactId>some-library</artifactId>
    <version>2.0.0</version>
    <exclusions>
        <exclusion>
            <groupId>com.conflict</groupId>
            <artifactId>conflicting-library</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Question 16: Ensuring a Clean Build for a Multi-Module Project

You are managing a multi-module Maven project, and you notice that sometimes old or generated files from previous builds interfere with the current build, causing failures. To prevent this, you need to ensure that a clean operation is always performed before any new build.

How would you ensure that the clean phase is executed every time before a new build in a multi-module project? Which command would you run to ensure a clean build across all modules? To ensure the `clean` phase is executed before every build in a multi-module project, you can run the following command:

```
mvn clean install
```

This command ensures that the `clean` phase is executed for all modules before the `install` phase.

Question 17: Adding a Custom Clean Goal

n addition to the standard cleaning of the target directory, your project generates temporary files in a custom folder (temp) outside of the target directory. You want to ensure that these

files are also deleted during the clean phase.

How would you customize the clean lifecycle to include the deletion of files from the temp directory? How would you configure the pom.xml to achieve this? To delete files from a custom folder (e.g., temp) during the `clean` phase, you can configure the `maven-clean-plugin`:

```xml
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-clean-plugin</artifactId>
            <version>3.1.0</version>
            <configuration>
                <filesets>
                    <fileset>
                        <directory>temp</directory>
                    </fileset>
                </filesets>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Question 18: Handling Dependencies that Generate Files

Your project has a dependency on a third-party library that generates temporary files in a specific directory (generated-files/). You need to make sure these files are cleaned up before the next build starts.

How would you ensure that the files generated by the third-party dependency are cleaned up when the clean lifecycle is triggered? What changes to the pom.xml would you make? To ensure that files generated by a third-party library are cleaned up, you can configure the `maven-clean-plugin` to remove the `generated-files` directory:

```xml
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-clean-plugin</artifactId>
            <configuration>
                <filesets>
                    <fileset>
                        <directory>generated-files</directory>
                    </fileset>
                </filesets>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Question 19: Automatically Cleaning Before Testing

You are working on a project where stale files from previous builds sometimes cause unit tests to fail. You want to ensure that every time the test phase is executed, a clean operation is automatically performed before it.

How would you modify the Maven lifecycle to ensure that the clean phase runs automatically before the test phase? How can you ensure that this happens without manually typing mvn clean test each time? To automatically run the `clean` phase before the test phase, you can bind the clean goal to the `pre-integration-test` phase in the POM:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-clean-plugin</artifactId>
            <executions>
                <execution>
                    <phase>pre-integration-test</phase>
                    <goals>
                        <goal>clean</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

This ensures that the clean phase runs before testing without manually typing `mvn clean test`.

Question 20: Customizing the Clean Phase with Additional Goals

In addition to cleaning the build directory, you need to perform additional cleanup actions such as clearing cache files and deleting logs that are stored outside the project structure (e.g., in the user's home directory)

How would you extend the clean lifecycle to include additional cleanup actions, such as clearing a cache directory and removing log files? To extend the clean phase and delete additional files (e.g., cache and logs outside the project), you can configure the `maven-antrun-plugin` in the clean phase:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-antrun-plugin</artifactId>
            <executions>
                <execution>
                    <phase>clean</phase>
                    <configuration>
                        <tasks>
                            <delete dir="${user.home}/cache" />
                            <delete file="${user.home}/logs/application.log" />
                        </tasks>
                    </configuration>
                    <goals>
                        <goal>run</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Question 21: Cleaning Dependent Projects in a Multi-Module Build

In a multi-module project, certain modules depend on each other. You want to ensure that the clean phase is executed across all dependent modules if one module is cleaned.

How would you ensure that cleaning one module in a multi-module project triggers the clean phase for all dependent modules? How would you configure the build to handle this situation? To ensure that cleaning one module triggers cleaning of its dependent modules, use the `-am` (also make) option in the Maven command:

```
mvn clean -pl module-a -am
```

This will clean module-a and all dependent modules.

Question 22: Managing Build Artifacts During the Clean Phase

Your project is using third-party libraries that download files into a local repository or external folders during the build process. However, you don't want these downloaded files to be removed during the clean phase because they are needed for future builds.

How would you exclude certain directories or files (e.g., the repository directory) from being deleted during the clean phase while still cleaning the main target directory? To exclude certain directories (like a repository directory) from being cleaned, configure the `maven-clean-plugin` as follows:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-clean-plugin</artifactId>
            <configuration>
                <excludeDefaultDirectories>true</excludeDefaultDirectories>
                <filesets>
                    <fileset>
                        <directory>${project.build.directory}</directory>
                    </fileset>
                </filesets>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Question 23: Reusing Common Build Configurations

You are managing a set of Maven projects, and they all share common configurations such as Java version, plugin configurations, and dependency versions. You want to centralize these configurations so that each child project does not need to repeat them.

How would you structure a parent POM file to manage shared configurations (such as Java version and commonly used plugins) that all child projects can inherit? What elements of the parent POM would you define for reuse? To centralize common configurations in a parent POM, you can define shared properties, plugin configurations, and dependencies in the parent POM:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>11</source>
                <target>11</target>
            </configuration>
        </plugin>
    </plugins>
</build>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>5.3.0</version>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Question 24: Overriding Parent Configurations in Child Projects

You have a parent POM that defines shared configurations, but one of your child projects requires a different Java version (Java 11 instead of Java 8) and an additional plugin. You want to override the Java version and add the new plugin in the child POM without affecting other child projects.

How would you override the Java version and add a new plugin in the child POM? What parts of the child POM would you need to modify? To override the Java version and add a new plugin in a child project, you can modify the child POM as follows:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>11</source>
                <target>11</target>
            </configuration>
        </plugin>
        <plugin>
            <groupId>new.plugin.group</groupId>
            <artifactId>new-plugin</artifactId>
            <version>1.0</version>
        </plugin>
    </plugins>
</build>
```

Question 25: Managing Dependencies in Parent and Child Projects

Your parent POM defines several common dependencies, but a child project requires a different version of one of these dependencies. You want to override the version of this dependency in the child project without modifying the parent POM.

How would you override the version of a dependency in a child project while inheriting other dependencies from the parent? What part of the child POM would you modify to achieve this?

To override the version of a dependency in the child project, use the `<dependencies>` section in the child POM:

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>5.2.0</version> <!-- Override version -->
    </dependency>
</dependencies>
```

Question 26: Using Profiles in Parent POM for Environment-Specific Configurations

You have a parent POM that defines different configurations for various environments (development, staging, production). Each child project should automatically inherit these configurations, and you want to use Maven profiles to switch between environments.

How would you set up environment-specific configurations using Maven profiles in the parent POM? How can the child projects inherit and use these profiles? In the parent POM, define profiles for different environments, and the child projects can inherit them:

```
<profiles>
    <profile>
        <id>development</id>
        <properties>
            <db.url>jdbc:dev-url</db.url>
        </properties>
    </profile>
    <profile>
        <id>production</id>
        <properties>
            <db.url>jdbc:prod-url</db.url>
        </properties>
    </profile>
</profiles>
```

Activate a profile during the build:

```
mvn clean install -Pproduction
```

Question 27: Inheriting and Extending Build Plugins

The parent POM defines several build plugins that are used across all projects. However, one of the child projects requires additional configuration for one of these plugins (e.g., a different goal or execution phase). You need to extend the plugin configuration in the child POM without affecting other projects.

How would you extend the configuration of an inherited plugin in the child POM? What changes would you make to the child POM to achieve this? To extend the configuration of an inherited plugin in the child POM, without affecting other projects, you can redefine the plugin in the child POM and specify the additional configuration or goals. Maven allows you to inherit the plugin configuration from the parent but also add new goals or modify the execution phase.

Example of extending a plugin:

Parent POM:

```
<build>
    <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>
</build>
```

Child POM:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <executions>
                <execution>
                    <id>custom-compile</id>
                    <phase>compile</phase>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                    <configuration>
                        <fork>true</fork>
                        <meminitial>512m</meminitial>
                        <maxmem>1024m</maxmem>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

In this case, the `maven-compiler-plugin` is inherited, but the child module adds a custom execution with additional memory configuration for the `compile` phase.

Question 27: Managing Parent-Child Relationships in a Multi-Module Project

You are working on a multi-module Maven project where each child module inherits from a common parent POM. However, one of the child modules has specific requirements that need a new plugin or different configuration that should not affect the other child modules. You want to ensure that only this child module includes the new plugin without altering the configurations in other modules.

How would you handle the situation where one child module requires a new plugin or different configuration, but this should not impact the other child modules? What changes would you make in the child POM to meet this requirement? If one child module requires a new plugin or different configuration, you can add the new plugin only in the child POM, while the other modules inherit from the parent POM as usual. The new plugin will only apply to that specific child module.

Parent POM:

```
<build>
    <plugins>
        <!-- Common plugins here -->
    </plugins>
</build>
```

Child POM (with specific plugin):

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.0.0-M5</version>
            <configuration>
                <includes>
                    <include>**/CustomTest*.java</include>
                </includes>
            </configuration>
        </plugin>
    </plugins>
</build>
```

This way, the `maven-surefire-plugin` is applied only to the child module, without affecting the rest of the multi-module project.

Question 28: Inheriting and Customizing Dependency Versions in Child Projects

Situation: You have a parent POM that defines common dependencies using . Some child projects need to use the same dependencies but with different versions. You need to ensure that child projects can override the versions without modifying the parent POM.

How would you allow child projects to inherit common dependencies from the parent POM but use different versions where needed? What changes would you make in the child POM to override the dependency versions? If you want the child project to inherit dependencies but override specific versions, you can use the `<dependencyManagement>` section in the parent POM, and then define the specific version of the dependency in the child POM.

Parent POM:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>5.3.0</version>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Child POM (with overridden version):

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>5.2.0</version> <!-- Override version -->
    </dependency>
</dependencies>
```

The child POM will inherit all other dependencies from the parent but will use version `5.2.0` of `spring-core` instead of `5.3.0`.

Question 29: Customizing the Clean Lifecycle

You have a multi-module project, and one of the child modules requires a custom action during the clean phase (such as deleting specific files that are not included in the standard Maven clean). The other child modules should retain the default behavior of the clean phase.

How would you customize the Maven clean phase for a specific module while retaining the default clean behavior for other modules? What changes would you make in the child POM to customize the clean phase? To customize the clean phase for one specific module, while the other modules retain the default behavior, you can configure the `maven-clean-plugin` in the child POM.

Child POM (custom clean):

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-clean-plugin</artifactId>
            <executions>
                <execution>
                    <phase>clean</phase>
                    <goals>
                        <goal>clean</goal>
                    </goals>
                    <configuration>
                        <filesets>
                            <fileset>
                                <directory>${basedir}/custom-temp</directory>
                            </fileset>
                        </filesets>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

This configuration ensures that during the clean phase, the `custom-temp` directory is cleaned in this specific module, while other modules continue with the default clean behavior.

Question 30: Using Plugin Inheritance with Custom Execution Goals

The parent POM defines several common plugins used by all child modules. One of the child modules, however, requires an additional goal for one of these plugins (e.g., running tests with a specific configuration).

How would you extend the plugin configuration in the child POM to add a custom execution goal for a plugin that is inherited from the parent POM? What changes would you make to the child POM to include the additional goal? To add a custom execution goal for an inherited plugin, you can extend the plugin configuration in the child POM. Here's how you can define an additional execution goal:

Parent POM:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.0.0-M5</version>
        </plugin>
    </plugins>
</build>
```

Child POM (with custom goal):

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <executions>
                <execution>
                    <id>custom-tests</id>
                    <phase>test</phase>
                    <goals>
                        <goal>test</goal>
                    </goals>
                    <configuration>
                        <includes>
                            <include>**/SpecialTest*.java</include>
                        </includes>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

In this example, the `maven-surefire-plugin` is inherited from the parent POM, but the child project adds a custom execution for running only specific tests. This does not affect other modules.