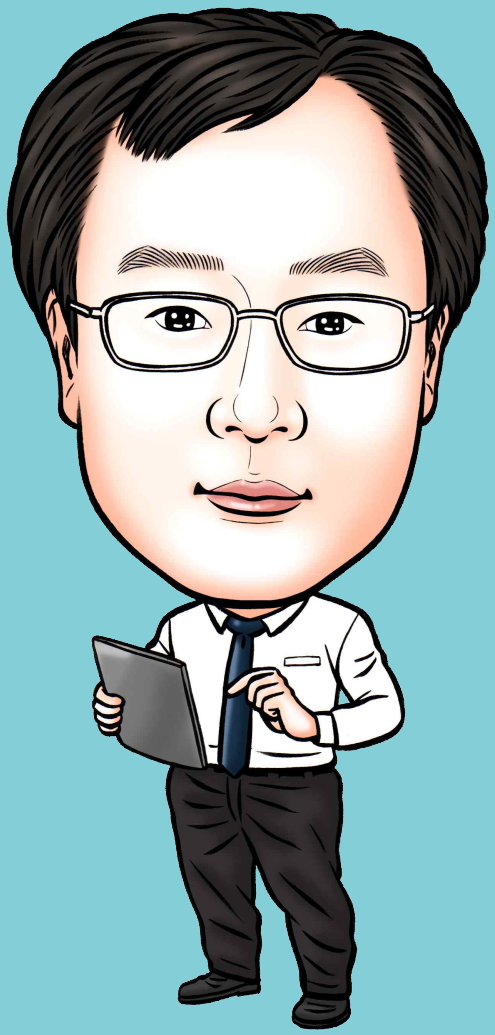
PART 1. 스프링의 메이븐



PART

목차

STEP 01

메이븐의 소개

메이븐의 이해

메이븐의 개요

의존성을 관리하는 빌드 툴인 메이븐(Maven)은 자바 개발자가 자주 사용하는 빌드 툴 중 하나다.
빌드는 소스 코드를 실행할 수 있는 소프트웨어 아티팩트(Artifact)로 변환하는 과정이다.
빌드 프로세스를 통해 생성된 실행 가능한 JAR 파일, WAR 파일 등의 파일이나 모듈이 아티팩트다.
아티팩트는 원본 소스 코드를 컴파일하고 패키지화한 결과물로서 실제로 실행되거나 배포될 수 있다.
프로젝트 구조와 내용을 기술하는 선언적 접근 방식의 오픈 소스 빌드 툴로 일반적으로 다음 단계를 포함한다.

- 컴파일 : 소스 코드가 컴퓨터가 이해하고 실행할 수 있는 바이너리 코드로 변환되는 과정이다.
- 테스트 : 자동 테스트를 실행하여 코드에 오류가 없는지 확인한다.
- 패키징 : 컴파일된 코드를 특정 형식인 JAR 파일, WAR 파일 등으로 묶는다.
- 설치배포 : 패키징된 소프트웨어를 특정 환경에 설치하거나 배포한다.

플러그인을 구동해주는 프레임워크로 모든 작업은 플러그인에서 수행한다.
테스트를 병행하거나 서버 측의 배치 리소스를 관리할 수 있는 환경을 제공한다.
모든 프로젝트가 일관된 디렉터리 구조와 빌드 프로세스 유지하며 다양한 플러그인 사용이 가능하다.
다음과 같은 기능을 메이븐에서 제공하고 있다.

- 빌드 : 프로젝트를 컴파일, 테스트, 패키징하고 설치하는 빌드 과정을 자동화한다.
- 문서화 : 프로젝트에 대한 문서를 생성하고 관리하는 기능을 제공한다.
- 리포팅 : 프로젝트의 진행 상황, 코드 품질, 테스트 결과 등에 대한 정보를 제공한다.
- 의존성 관리 : 프로젝트의 모든 라이브러리와 플러그인의 종속성을 관리한다.
- 코드 관리 : 소스 코드 관리 도구와 통합되어 코드 변경 내용을 추적하고 버전 관리를 쉽게 할 수 있다.
- 릴리즈 관리 : 프로젝트의 버전을 관리하고 새로운 버전의 프로젝트를 릴리즈하는 과정을 자동화한다.
- 배포 : 생성된 아티팩트를 원격 저장소에 배포하는 기능을 제공한다.

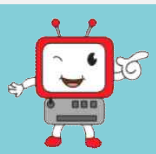
메이븐의 장단점

장점

모든 프로젝트가 일관된 디렉터리 구조와 빌드 프로세스를 유지할 수 있다.
표준화된 프로젝트 구조와 빌드 명령을 제공하므로 관리를 일관되게 할 수 있다.
중앙 저장소를 통해 다양한 라이브러리를 편리하게 관리하고 사용할 수 있다.
컴파일, 테스팅, 보고, 문서화 등 다양한 작업을 수행할 수 있는 플러그인을 제공한다.
아키타입(Archetype) 기능으로 프로젝트의 기본 구조를 정의하는 템플릿을 제공한다.

단점

메이븐의 사용 방법과 프로젝트 설정 방법을 배우는 데 시간이 걸릴 수 있다.
pom.xml 파일 구성과 플러그인 시스템은 초보자에게 복잡할 수 있다.
표준화된 프로젝트 구조는 특정 프로젝트 요구사항에 대한 유연성을 제한할 수 있다.

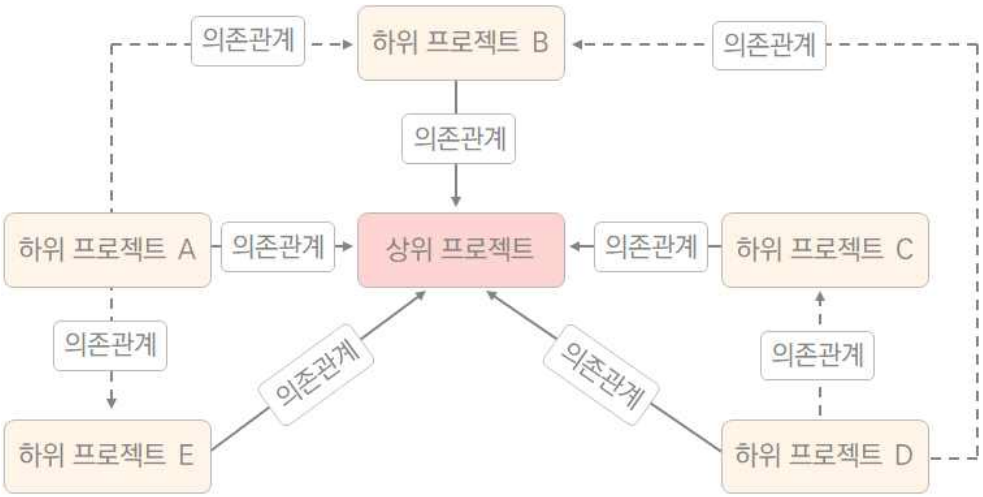


1.메이븐의 소개

■ 메이븐의 관리

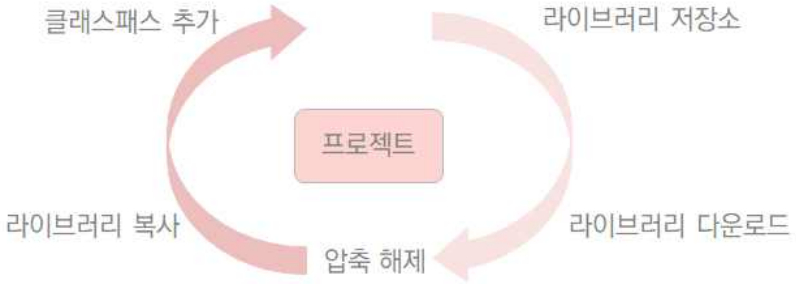
✔ 프로젝트

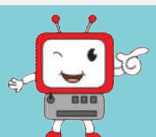
프로젝트를 분리할 때는 하위 프로젝트가 상위 프로젝트에 대해서만 의존관계를 맺는 것이 원칙이다.
프로젝트를 진행하면 원칙에 벗어나서 하위 프로젝트 간에도 의존관계가 발생할 수 있다.
프로젝트의 복잡도가 증가하면은 빌드의 복잡도 또한 증가한다.
빌드에 대한 관리가 제대로 되지 않으면 빌드가 깨지는 경우가 자주 생기고 빌드 시간이 길어질 수도 있다.
메이븐으로 빌드를 관리하면 빌드의 문제점을 해결할 수 있으므로 빌드에 대한 효율성을 높일 수 있다.



✔ 라이브러리

의존관계에 있는 라이브러리가 다른 외부 라이브러리와 다시 의존관계를 맺으면 프로젝트가 매우 복잡해질 수 있다.
외부 라이브러리가 점점 더 많아지고 복잡해지고 있으며 라이브러리 관리 방법에도 한계를 가질 수밖에 없다.
메이븐은 복잡한 라이브러리의 의존관계를 자동으로 빌드하여 라이브러리에 대한 의존관계를 효율적으로 관리한다.
라이브러리 관리에 필요한 모든 작업을 추상화하고 표준화해서 반복을 제거한다.
개발자에게 메이븐의 최대 장점은 프로젝트의 라이브러리와 종속 라이브러리에 영향을 미치는 디펜던시(dependency) 리소스까지 자동으로 관리 할 수 있다는 점이다.
라이브러리 파일인 jar 파일을 내려받아 프로젝트에 추가할 때 연관된 다른 종속 라이브러리를 추가해야 하는 불편함을 해소할 수 있다.
편리하고 일관성 있는 라이브러리 간의 의존관계 관리뿐만 아니라 프로젝트별 모듈의 의존성도 관리할 수 있다.
프로젝트 전반의 리소스 관리, 표준 디렉터리 구조 등을 처음부터 일관된 형태로 구성한다.
메이븐은 라이브러리가 접근할 수 있는 저장소를 지원한다.
저장소를 통하여 템플릿 프로젝트인 아키타입(Archetype), 플러그인 기능, 의존관계에 있는 라이브러리 등이 지원된다.
의존 라이브러리는 저장소에서 프로젝트를 빌드하는 시점에 개발자 PC에 자동으로 내려받을 수 있다.





1.메이븐의 소개

메이븐의 지원

아키타입(Archetype)

아키타입은 프로젝트의 기본 구조를 빠르게 설정할 수 있는 템플릿을 제공한다.
템플릿으로 프로젝트를 생성하면 아키타입 기능으로 프로젝트의 뼈대를 자동 생성한다.
아키타입은 프로젝트 구조의 원형이나 모델을 의미한다.
개발자는 표준 디렉터리 구조, 예제 코드, 설정 파일 등을 미리 정의된 형태로 쉽게 생성할 수 있다.
프로젝트 간에 일관성을 유지하는 데 도움이 되며 아키타입의 디렉터리는 다음과 같다.

- src/main/java 디렉터리 : 자바 소스 파일 위치로 하위에 패키지를 배치한다.
- src/main/resources 디렉터리 : XML 파일이나 properties 파일 등의 리소스 파일이 위치한다.
- src/test/java 디렉터리 : JUnit 등의 단위 테스트 파일이 위치한다.
- src/test/resources 디렉터리 : 단위 테스트에 필요한 XML 파일이나 properties 파일 등의 리소스 파일이 위치한다.
- src/main/webapp 디렉터리 : 웹 애플리케이션의 서버 파일이 위치한다.
- src/main/webapp/resources 디렉터리 : 웹 애플리케이션의 웹 브라우저 파일인 css 파일, js 파일, img 파일 등이 위치한다.
- target 디렉터리 : 컴파일 시에 필요한 리소스 파일이 위치한다.

라이브러리(Library)

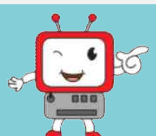
POM 설정 파일인 pom.xml의 <dependencies> 태그를 사용하여 라이브러리를 관리한다.
<dependencies> 태그의 하위에 <dependency> 태그로 라이브러리를 등록하거나 변경한다.
기존의 Junit 라이브러리를 설정한 <dependency> 태그에서 예시와 같이 변경 후 단위 테스트를 실행할 수 있다.

예시

```
<!--프로젝트에 의존관계가 있는 라이브러리를 관리한다.-->
<dependencies>

  <!--의존관계가 있는 라이브러리를 등록하여 다운로드한다.-->
  <dependency>

    <!--라이브러리의 고유 아이디를 설정한다.-->
    <groupId>junit</groupId>

    <!--라이브러리를 식별하는 유일한 아이디를 설정한다.-->
    <artifactId>junit</artifactId>

    <!--Junit 버전을 설정한다.-->
    <version>5.10.2</version>

    <!--라이브러리의 범위를 설정하며 test 스코프는 최종 배포 산출물을 포함하는 시점에는 포함되지 않는다.-->
    <scope>test</scope>

  </dependency>
</dependencies>
```

라이브러리의 실행 흐름은 다음과 같다.

1. <dependencies> 태그로 중앙 저장소에 접근한다.
2. 하위 태그인 <dependency> 태그에서 필요한 라이브러리를 선택하여 다운로드한다.
3. 선택한 라이브러리의 모든 의존성 라이브러리가 함께 다운로드 된다.

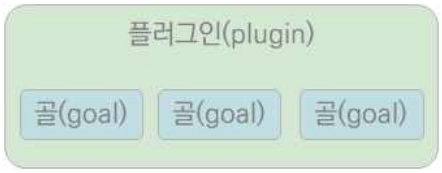


1.메이븐의 소개

■ 플러그인(plugin)

✔ 플러그인의 소개

추가 기능인 플러그인은 호스트 응용 프로그램과 서로 응답하는 컴퓨터 프로그램을 의미한다.
플러그인은 특정 소프트웨어의 기능을 확장하거나 추가하는 데 사용된다.
메이븐은 다수의 플러그인으로 구성되어 있으며 각각의 플러그인은 하나 이상의 작업인 골(goal)을 포함하고 있다.



골은 빌드 프로세스의 특정 단계를 나타낸다.
각 골은 특정 작업을 수행하도록 프로그램되어 있다.
골은 메이븐 빌드 라이프사이클의 특정 단계에 바인딩될 수 있다.
빌드 프로세스를 자동화하면서 각 단계에서 적절한 작업을 실행할 수 있다.
골은 직접 명령어를 통해 호출할 수도 있다.
골을 수행하는 데 필요한 모든 전처리 작업을 자동으로 수행한다.
메이븐의 기본 기능 외에 추가로 플러그인을 만들어서 미리 정의된 빌드 과정 중에 부가 기능으로 추가할 수 있다.
플러그인은 메이븐의 중앙 저장소에서 찾아서 사용할 수 있다.
메이븐의 중앙 저장소에서 메이븐의 핵심 플러그인 대부분을 검색하여 설치할 수 있다.
빌드의 포괄적인 재사용을 위하여 플러그인 모두를 저장소에서 검색하여 내려받을 수 있도록 허용한다.
공통의 빌드 작업을 매번 새로 작성하지 않아도 된다.
설치된 메이븐의 업그레이드는 POM 설정 파일에서 플러그인 버전만 변경하면 된다.
메이븐 플러그인을 통하여 전역적으로 재사용할 수 있다.
빌드와 배포 과정을 자동화하고 확장하는 데 필요한 기능을 제공하며 특징은 다음과 같다.

■ 재사용 가능

다수의 프로젝트 간에 재사용할 수 있다.
공통된 빌드나 테스트 요구사항을 일관되게 관리할 수 있다.

■ 확장 가능

필요에 따라 사용자 정의 플러그인을 작성하여 기능을 확장할 수 있다.

■ 생명 주기 관리

빌드 생명 주기의 다양한 단계에 대한 작업을 제공한다.

■ 환경 설정

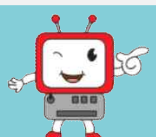
pom.xml 파일 내에 설정할 수 있으며 플러그인의 동작을 프로젝트의 요구사항에 맞게 커스터마이징할 수 있다.

■ 자동 의존성 관리

메이븐의 중앙 저장소를 통해 다운로드된다.
플러그인이 필요로 하는 추가적인 라이브러리도 자동으로 관리된다.

■ 표준화

프로젝트 빌드를 표준화하고 일관성을 유지하는 데 도움이 된다.
다양한 프로젝트와 팀 간에 빌드 과정이나 배포 과정의 통일성을 보장하는 데 중요하다.



1.메이븐의 소개

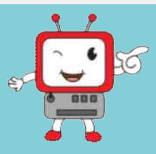
플러그인의 변경

메이븐의 기본 빌드를 커스터마이징(customizing) 할 때 주로 메이븐 기본 플러그인을 설정한다.
커스터마이징은 개발자가 사용 방법과 기호에 맞춰 소프트웨어를 설정하거나 기능을 변경하는 것이다.
기본 플러그인은 메이븐의 핵심 기능을 제공하며 메이븐 빌드 라이프사이클의 각 단계를 처리한다.
메이븐을 업데이트할 때 프로젝트에서 변경한 환경으로 유지가 되지 않고 기본 컴파일 플러그인으로 복구된다.
기본 컴파일 플러그인으로 복구되지 않도록 <build> 태그의 하위 태그인 <plugins> 태그에서 변경할 수 있다.
<plugins> 태그 안에서 원하는 플러그인을 지정한다.
지정된 플러그인의 동작을 변경하거나 추가하는데 필요한 설정을 제공할 수 있다.

예시

```
<!--플러그인을 빌드한다.-->
<build>

  <!--메이븐 플러그인을 관리한다.-->
  <plugins>

    <!--메이븐 플러그인을 등록하여 다운로드한다.-->
    <plugin>

      <!--플러그인의 고유 아이디를 설정한다.-->
      <groupId>org.apache.maven.plugins</groupId>

      <!--플러그인을 식별하는 유일한 아이디를 설정한다.-->
      <artifactId>maven-compiler-plugin</artifactId>

      <!--플러그인의 버전을 설정한다.-->
      <version>3.11.0</version>

      <!--컴파일 환경을 설정한다.-->
      <configuration>

        <!--컴파일되는 소스 파일의 JDK 버전을 설정한다.-->
        <source>17</source>

        <!--클래스 파일이 동작하게 될 JDK 버전을 설정한다.-->
        <target>17</target>

        <!--컴파일되는 소스 파일의 인코딩을 설정한다.-->
        <encoding>UTF-8</encoding>

      </configuration>

    </plugin>

  </plugins>

</build>
```

maven-compiler-plugin의 설정을 변경하여 자바 소스 코드와 바이트 코드의 버전을 변경한다.
자바 소스 코드와 바이트 코드를 상속으로 등록하려면 ${java.version} 형식으로 설정하면 된다.
소스 파일의 인코딩을 상속으로 등록하려면 ${project.build.sourceEncoding} 형식으로 설정하면 된다.
메이븐을 업데이트해도 변경된 설정은 보존된다.
메이븐 업데이트 후에도 프로젝트의 빌드 설정이 변경되지 않도록 할 수 있다.



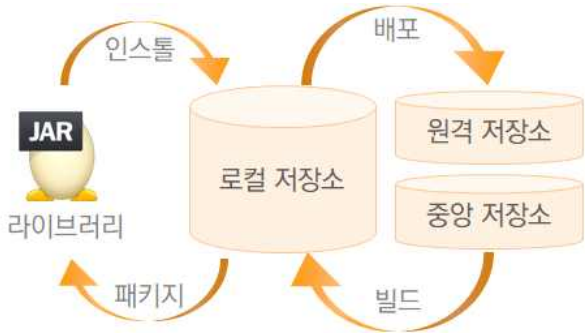
STEP 02

메이븐의 저장소

저장소의 소개

저장소의 이해

저장소는 의존관계 라이브러리와 플러그인을 보관하고 있는 공간이다.
POM 설정 파일인 pom.xml의 <repositories> 태그에서 메이븐의 중앙 저장소에 대한 정보만 설정되어 있다.
중앙 저장소가 아닌 다른 저장소에서 제공하는 라이브러리는 <repositories> 태그 내에 설정을 추가해 주어야 한다.
메이븐이 라이브러리를 내려받을 때는 <repositories> 태그에 설정된 저장소 순서로 진행한다.
중앙 저장소에서 의존관계에 있는 라이브러리를 검색하여 찾았다면 다음 저장소에 접근하지 않는다.



저장소의 형태

중앙 저장소

메이븐에서 운영하는 저장소다.
개발자가 임의로 라이브러리를 배포할 수 없다.
오픈 소스 라이브러리, 메이븐 플러그인, 메이븐 아키타입을 관리한다.

원격 저장소

메이븐이 아닌 다른 개발 회사에서 운영하는 저장소다.
개발자가 임의로 라이브러리를 배포할 수 없다.

- 사내 원격 저장소 : 자체 회사 내에서만 사용하기 위해 제공하는 저장소다.
- 공개 원격 저장소 : 스프링, 오라클, 자카르타와 같은 프로그램 회사에서 제공하는 저장소다.

로컬 저장소

개발자의 컴퓨터에서 운영하는 저장소로 개발자가 임의로 라이브러리를 배포할 수 있다.
중앙 저장소나 원격 저장소에서 다운로드한 라이브러리, 플러그인을 관리한다.
운영체제에서 기본적으로 제공하는 로컬 저장소 위치는 운영체제에 따라 다르다.

- 윈도 : C:₩Users₩${userid}
- 리눅스 : /home/${userid}
- Mac : /usr/${userid}

운영체제에서 제공하는 로컬 저장소의 위치는 필요에 따라서 개발자가 임의로 변경할 수 있다.

2.메이븐의 저장소

저장소의 설정

원격 저장소 설정

대부분의 의존 라이브러리는 중앙 저장소에서 제공지만, 일부 라이브러리는 별도의 원격 저장소를 통하여 제공한다.
메이븐은 필요한 아티팩트를 찾기 위해 원격 저장소를 사용한다.
로컬 저장소에서 필요한 아티팩트를 찾지 못하면 설정된 원격 저장소를 차례로 검색하여 아티팩트를 찾는다.
특정 조직이나 커뮤니티에서 제공하는 아티팩트에 접근하거나 고유한 아티팩트를 공유하는 데 사용할 수 있다.
원격 저장소는 웹 서버에 의존하므로 서버 문제나 저장소 자체가 중단될 때 접근할 수 없게 될 수 있다.
안정적인 빌드를 위해서는 사용하는 중요한 종속성을 로컬 저장소에 수동으로 등록하는 것을 권장한다.
빌드 속도를 향상하며 네트워크 문제나 원격 저장소의 문제로 인한 빌드 실패를 방지할 수 있다.
별도의 원격 저장소에서 라이브러리를 다운로드하려면 POM 설정 파일인 pom.xml에서 <repositories> 태그의 하위 태그인 <repository> 태그에서 원격 저장소를 설정한다.

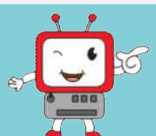
예시

```
<!--의존성 라이브러리의 원격 저장소를 관리한다.-->
<repositories>

  <!--의존성 라이브러리의 원격 저장소를 등록한다.-->
  <repository>

    <!--원격 저장소의 식별자를 설정한다.-->
    <id>image</id>

    <!--원격 저장소의 이름을 설정한다.-->
    <name>Thumbnail Repository</name>

    <!--네트워크 안에 있는 원격 저장소의 경로를 설정하면 해당 저장소에서 로컬 저장소 경로로 내려받는다.-->
    <url>http://maven.geotoolkit.org/</url>

  </repository>
</repositories>
```

원격 저장소를 설정하고 나서 <dependencies> 태그의 하위 태그인 <dependency> 태그에서 등록하여 다운로드한다.

예시

```
<!--프로젝트에 의존관계가 있는 라이브러리를 관리한다.-->
<dependencies>

  <!--의존관계가 있는 라이브러리를 등록하여 다운로드한다.-->
  <dependency>

    <!--라이브러리의 고유 아이디를 설정한다.-->
    <groupId>net.coobird</groupId>

    <!--라이브러리를 식별하는 유일한 아이디를 설정한다.-->
    <artifactId>thumbnailator</artifactId>

    <!--thumbnailator 버전을 설정한다.-->
    <version>0.4.8</version>

  </dependency>
</dependencies>
```



2.메이븐의 저장소

로컬 저장소 설정

형식

mvn install:install-file -Dfile=라이브러리 경로 -DgroupId=groupId 이름
-DartifactId=artifactId 이름 -Dversion=version 이름 -Dpackaging=패키징 형태

라이브러리 버전이 충돌할 가능성을 최소화할 수 있으며 빌드를 처음 할 때 필요한 수동적인 작업을 줄일 수 있다.
한 번만 배포하면 되며 초기 빌드 시간을 상당히 단축할 수 있고 중앙 저장소에 없는 라이브러리를 수동으로 등록할 수 있다.
독립적으로 관리하는 로컬 저장소에서 의존관계가 있는 모든 라이브러리를 관리할 수 있다.
groupId는 일반적으로 도메인명을 사용하며 groupId에 도메인명을 사용할 때 .(도트)를 기준으로 디렉터리를 분리해 관리한다.
메이븐 명령문에서 D는 define의 약자로 생각하면 된다.
라이브러리 버전에 대한 정보 확인은 다운로드한 jar 파일에 대한 압축을 해제하고 압축이 해제된 디렉터리 중에서 META-INF 디렉터리에 있는 MANIFEST.MF 파일을 메모장으로 열어서 확인할 수 있다.
외부 메이븐을 설치한 경우에는 관리자 명령 프롬프트에서 메이븐 명령어로 빌드하고 인텔리제이의 내장 메이븐을 사용하는 경우에는 인텔리제이의 터미널에서 동일한 메이븐 명령어로 빌드할 수 있다.

메이븐 빌드 실행

명령 프롬프트에서 mvn install:install-file 명령어로 빌드를 실행한다.

예시

관리자: 명령 프롬프트

```
Microsoft Windows [Version 10.0.18363.1139]
(c) 2019 Microsoft Corporation. All rights reserved.

C:₩WINDOWS₩system32>mvn install:install-file -Dfile=C:₩library₩ojdbc11.jar -DgroupId=com.oracle.
database.jdbc -DartifactId=ojdbc11 -Dversion=23.8.0.25.04 -Dpackaging=jar
```

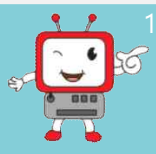
메이븐 빌드 완료

빌드에 대한 BUILD SUCCESS 메시지가 출력하면 빌드가 성공한 것이고 BUILD FAILURE 메시지가 출력하면 실패한 것이므로 다시 빌드해야 한다.

예시

관리자: 명령 프롬프트

```
Microsoft Windows [Version 10.0.18363.1139]
(c) 2019 Microsoft Corporation. All rights reserved.

C:₩WINDOWS₩system32>mvn install:install-file -Dfile=C:₩library₩ojdbc11.jar -DgroupId=com.oracle.
database.jdbc -DartifactId=ojdbc11 -Dversion=23.8.0.25.04 -Dpackaging=jar
[INFO] Scanning for projects...
                :
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
C:₩WINDOWS₩system32>
```



STEP 03

메이븐의 빌드

빌드의 소개

빌드의 이해

빌드(Build)는 소스 코드 파일을 컴퓨터에서 실행할 수 있는 독립 소프트웨어 가공물이다.
소스 코드를 실행할 수 있는 소프트웨어 아티팩트(Artifact)로 변환하는 과정이나 결과물이다.
일반적으로 컴파일, 테스트, 패키징, 배포를 포함한다.
소스 코드의 구문 검사, 바이너리 코드로의 변환, 필요한 라이브러리와의 연결 등이 포함될 수 있다.
아티팩트는 빌드 과정을 통해 생성된 결과물을 가리킨다.
결과물은 JAR, WAR, EAR와 같은 실행 가능한 파일 형식으로 생성된다.
빌드에 있어 가장 중요한 단계는 소스 코드 파일이 실행 코드로 변환되는 컴파일 과정이다.
컴파일은 소스 코드 파일을 실행 가능한 코드인 바이너리 코드로 변환한다.
컴파일러는 소스 코드의 문법을 확인하고 오류를 발견하면 개발자에게 알려준다.
컴파일 단계 이후에 테스트, 패키징, 배포 등의 단계가 이어진다.

빌드의 지원

코드 컴파일

테스트를 포함한 소스 코드를 컴파일한다.

컴포넌트 패키징

컴파일된 코드와 필요한 리소스들을 하나의 배포 가능한 패키지로 묶는다.
자바에서는 패키지를 JAR(Java Archive), WAR(Web Application Archive), EAR(Enterprise Archive) 등의 형태로 만든다.

- JAR : 자바 클래스와 관련된 메타데이터와 리소스를 패키징하며 확장자는 jar이다.
- WAR : 웹 애플리케이션을 패키징하며 확장자는 war이다.
- EAR : Java EE 애플리케이션을 패키징되며 확장자는 ear이다.

파일 조작

파일과 디렉터리를 만들고 복사하고 지우는 작업이다.

개발 테스트

자동화된 테스트를 진행한다.

버전 관리 도구 통합

버전 관리 시스템을 지원한다.

문서 생성

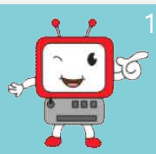
API 문서를 생성한다.

배포 기능

테스트 서버에서 배포를 지원한다.

코드 품질 분석

자동화된 검사 도구를 통한 코드 품질 분석을 한다.



3.메이븐의 빌드

빌드의 라이프사이클

라이프사이클의 소개

라이프사이클(LifeCycle)은 미리 정의한 빌드 순서로 빌드 프로세스의 특정 단계를 순서대로 배열한 것이다.
대부분 책임을 메이븐 플러그인에 위임하도록 설계되어 있으며 플러그인은 라이프사이클에 영향을 준다.
플러그인은 컴파일, 테스트 실행, 패키징, 문서 생성, 리포트 작성 등 다양한 작업을 수행한다.
일반적인 프로젝트 빌드 과정과 비슷하지만, 메이븐에서 다른 점이 있다면 빌드 단계를 미리 정하고 있다는 점이다.
소스 코드의 컴파일, 배포를 위한 패키징 등과 같은 것이 프로젝트의 프로세스다.
프로젝트의 프로세스를 메이븐의 핵심 플러그인에 적용하기 위하여 관습에 따른 구성을 사용한다.
아무런 설정을 추가하지 않고도 메이븐에서 제공하는 기본적인 프로젝트 빌드 기능을 사용할 수 있다.
모든 프로젝트가 일관된 디렉터리 구조와 빌드 프로세스를 유지하게 한다.
프로젝트의 기본 구조와 빌드 프로세스는 메이븐의 기본 규약을 바탕으로 구체화한다.
기본 규약으로 정의된 절차가 있으므로 마치 미리 정의해 둔 작업을 실행하듯이 사용한다.
정해진 디렉터리에 맞는 코드를 설정하면 메이븐이 알아서 해결하므로 개발자가 해야 할 일이 거의 없다.

라이프사이클의 요소

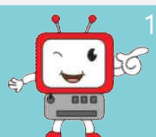
골(Goal)

골은 특정 작업을 수행하는 메이븐 플러그인의 명령으로 플러그인이 수행하는 작업 단위다.
플러그인에서 실행할 수 있는 각각의 작업이 골이며 특정 작업을 수행하는 일련의 골을 플러그인은 포함하고 있다.
포함된 골은 메이븐 라이프사이클의 특정 페이즈에 바인딩될 수 있다.
실질적인 빌드 작업은 각 페이즈에 연결된 골이 담당한다.
compile 페이즈와 연결된 compile:compile 골이 실행되면서 컴파일 작업을 진행한다.

해당 페이즈가 포함하고 있는 모든 페이즈에 등록된 골을 순차적으로 실행된다.
package 페이즈는 compile:compile 골 ➡ surefire:test 골 ➡ jar:jar 골을 순차적으로 실행한다.
각 페이즈를 실행할 때 기본으로 연결된 골을 실행하는 구조로 동작한다.
골이 실행되면 유닛 테스트(Unit Test)가 수행된다.
유닛 테스트는 소프트웨어의 가장 작은 단위로 일반적으로 함수나 메서드를 테스트하는 프로세스다.

페이즈(Phase)

라이프사이클의 빌드 단계가 페이즈이며 라이프사이클은 다수의 페이즈로 구성되어 있다.
페이즈를 기반으로 빌드를 실행하며 페이즈는 빌드 단계와 각 단계의 순서만을 설정한다.
빌드를 실행하기 위해서는 페이즈와 골로 빌드를 실행하며 페이즈가 빌드 작업은 하지는 않는다.
페이즈는 하나 이상의 골에 바인딩될 수 있으며 플러그인의 골을 실행시킬 수 있도록 접근을 허용한다.
각 페이즈는 빌드 과정의 특정 단계를 나타내며 각 페이즈는 순차적으로 실행되면서 빌드가 완성된다.
compile 페이즈의 작업이 성공적으로 완료되어야 test 페이즈로 넘어갈 수 있다.
먼저 테스트하려는 대상 소스 코드를 컴파일하고 컴파일 에러가 없으면 테스트 코드를 실행하는 것이다.
모든 페이즈가 순차적으로 실행되어 최종적으로 빌드가 완성되는 것이다.



3.메이븐의 빌드

■ 라이프사이클의 형태

build 라이프사이클

실제 소프트웨어 빌드를 수행하며 default 라이프사이클이라고도 한다.

컴파일, 테스트, 패키징, 설치 등의 페이즈를 포함한다.

■ compile 페이즈

compile 페이즈를 실행하면 먼저 의존관계에 있는 process-resources 페이즈가 실행된다.

process-resources 페이즈는 src/main/resources 디렉터리에 있는 리소스를 target/classes 디렉터리로 복사한다.

<resources> 태그를 재정의했을 경우 재정의한 규칙에 따라 리소스를 복사한다.

■ test-compile 페이즈

test-compile 페이즈를 실행하면 compile 페이즈를 먼저 실행된다.

compile 페이즈와 같이 코드를 컴파일하기 전에 process-test-resources 페이즈를 실행한다.

process-test-resources 페이즈를 실행하면 src/test/resources 디렉터리의 리소스를 먼저 복사한 후 테스트 코드를 컴파일한다.

src/test/java 디렉터리에 있는 테스트 코드는 src/main/java 디렉터리에 있는 코드와 의존관계를 맺는다.

코드를 컴파일하지 않은 상태에서 테스트 코드는 컴파일할 수 없다.

■ test 페이즈

JUnit 등으로 단위 테스트를 하며 기본적으로 단위 테스트가 실패하면 빌드 실패로 간주한다.

target/test-classes 디렉터리에 컴파일한 단위 테스트 클래스를 실행한다.

결과물을 target/surefire-reports 디렉터리에 생성한다.

test 페이즈는 test-compile 페이즈에 의존관계를 맺는다.

단위 테스트를 실행하지 않도록 설정할 때 일부 단위 테스트가 실패하더라도 다음 단계의 빌드를 실행하도록 설정한다.

■ package 페이즈

단위 테스트가 성공하면 POM 설정 파일의 <packaging> 태그의 속성값인 jar, war 등에 따라 압축한다.

compile, test-compile, test, package 순으로 실행된 후에 jar, war 파일 등이 target 디렉터리 하위에 생성된다.

기본적으로 <build> 태그의 하위 태그인 <finalName> 태그에 대한 값이 설정되어 있다면 ${finalName}.${packaging} 형태로 압축파일이 생성된다.

<finalName> 태그에 값이 설정되어 있지 않으면 ${artifactId}-${version}.${packaging}이 압축파일과 디렉터리 이름이 된다.

■ install 페이즈

로컬 저장소에 압축한 파일을 배포한다.

프로젝트를 빌드하여 생성된 아티팩트인 JAR, WAR 파일 등을 로컬 저장소에 설치한다.

jar, war 파일 등을 로컬 저장소에 등록하는 역할을 한다.

아티팩트는 다른 메이븐 프로젝트에서 의존성으로 사용될 수 있다.

install 페이즈는 package 페이즈와 의존관계를 맺는다.

■ deploy 페이즈

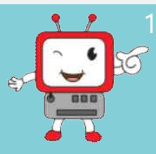
원격 저장소에 압축한 파일을 배포한다.

프로젝트를 빌드하여 생성된 아티팩트인 JAR, WAR 파일 등을 로컬 저장소에 설치한다.

jar, war 파일 등을 원격 저장소에 등록하는 역할을 한다.

아티팩트는 다른 메이븐 프로젝트에서 의존성으로 사용될 수 있다.

deploy 페이즈는 install 페이즈와 의존관계를 맺는다.



3.메이븐의 빌드

clean 라이프사이클

이전에 빌드된 아티팩트와 다른 임시 파일을 제거한다.

target 디렉터리의 결과물을 모두 제거하고 처음부터 새롭게 빌드를 하지만, 설치한 작업은 제거하지 않는다.

다른 페이즈와 의존관계가 없으므로 다른 페이즈를 실행해도 실행되지 않는다.

clean 페이즈를 실행하지 않으면 이전에 빌드 했던 결과물 중에서 불필요한 내용이 남아 있어 에러가 발생할 수 있다.

다른 페이즈를 실행할 때 반드시 clean 페이즈를 실행하고 빌드하는 습관을 지니는 것이 좋다.

프로젝트를 깨끗한 상태로 유지하고 이전 빌드로부터 발생할 수 있는 문제를 방지하는 데 도움이 된다.

이전 빌드에서 생성된 오래된 아티팩트가 새로운 빌드를 방해하는 것을 막을 수 있다.

이전 빌드 한 결과물을 제거하며 세 가지 기본 페이즈로 구성된다.

- **pre-clean 페이즈**
  clean 페이즈가 실행되기 전에 필요한 준비 작업을 수행한다.
- **clean 페이즈**
  이전에 빌드된 결과물을 제거하며 mvn clean 명령을 통해 실행될 수 있다.
- **post-clean**
  clean 페이즈의 작업 후에 추가적인 작업을 수행한다.

site 라이프사이클

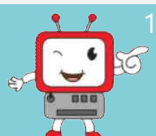
프로젝트 문서화와 관련된 작업을 수행하는 데 사용되며 다음과 같은 페이즈를 포함한다.

- **pre-site 페이즈**
  site 페이즈 생성 전 필요한 작업을 수행한다.
- **site 페이즈**
  프로젝트에 대한 문서를 생성하며 mvn site 명령을 통해 실행될 수 있다.
- **post-site 페이즈**
  site 페이즈 생성 후 추가 작업을 수행한다.
- **site-deploy 페이즈**
  생성된 문서를 특정 위치로 배포한다.

라이프사이클의 단계

플러그인을 사용하려면 POM 설정 파일인 pom.xml에서 <plugins> 태그를 사용하며 다음의 단계로 진행한다.

- **clean 단계**
  빌드된 결과물을 제거한다.
- **compile 단계**
  소스 코드를 컴파일한다.
- **test 단계**
  Junit 등의 정적분석 도구와 함께 단위 테스트를 수행한다.
- **package 단계**
  배포 가능한 형태인 jar, war 파일 등으로 컴파일 코드를 패키징 한다.
- **integration-test 단계**
  통합테스트를 진행하고 필요할 시 패키지를 배포한다.
- **install 단계**
  다른 프로젝트에서 종속 라이브러리로 사용될 수 있도록 패키지를 로컬 저장소에 설치한다.
- **deploy 단계**
  개발자나 프로젝트들과 공유할 수 있도록 최종 패키지를 원격 저장소에 복사한다.



STEP 04

메이븐의 의존 라이브러리

의존 라이브러리의 소개

의존 라이브러리의 이해

POM 설정 파일에 의존관계 설정 후에 테스트를 실행한다.
의존관계에 있는 라이브러리를 다운로드하고 빌드를 하며 프로젝트 하위 디렉터리가 아닌 로컬 저장소에 다운로드한다.
의존관계로 설정된 라이브러리를 프로젝트로 복사하는 dependency 플러그인을 제공한다.
dependency 플러그인 기능을 실행하면 디렉터리에 jar 파일을 복사한다.
디렉터리에는 POM 설정 파일에 추가한 jar 파일 외에 다른 jar 파일이 존재하는 것을 확인할 수 있다.
dependency 플러그인은 현재 프로젝트와 의존관계에 있는 라이브러리의 구조를 파악할 수 있도록 tree 골을 제공한다.
프로젝트가 추가될 때마다 해당 프레임워크와 의존관계에 있는 라이브러리를 추가해야 한다.
의존관계에 있는 라이브러리의 수가 증가하여 결과적으로 POM 설정 파일의 복잡도를 증가시킨다.
라이브러리를 다운로드하면 POM 설정 파일까지 재설정한다.
POM 설정 파일을 보면 해당 라이브러리와 의존관계에 있는 라이브러리 설정을 확인할 수 있다.

의존 라이브러리의 모듈

하나의 프로젝트에서 여러 모듈을 관리할 수 있는 기능을 지원한다.
한 프로젝트가 여러 모듈을 가지면서 빌드를 한 번에 진행하는 방법이다.

상속(Inheritance)

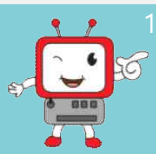
여러 프로젝트에 공통으로 적용되는 설정을 재사용할 수 있다.
부모 POM 파일을 생성하고 각 모듈에서 이를 상속받을 수 있다.
메이븐 설정 파일이 최상위 POM 설정 파일을 상속하듯이 프로젝트에서 공통으로 사용하는 설정을 지정한다.
부모인 공통 POM 설정 파일을 만들어 관리하고 하위 모듈에서 공통 POM 설정 파일을 상속할 수 있다.

집합(aggregation)

다수의 모듈을 하나의 프로젝트로 묶어서 관리하는 기능이다.
묶은 모듈을 한 번에 빌드하거나 테스트하는 등의 작업을 수행할 수 있다.
하나의 프로젝트가 여러 모듈로 분리될 때 모듈 간의 의존관계가 발생한다.
빌드를 모듈별로 진행할 때 빌드가 실패할 가능성이 크다.
각 모듈은 하나의 프로젝트로 같이 발전하기에 한 번에 빌드 할 필요가 있다.
여러 모듈을 빌드 할 때 같은 단위로 빌드 할 수 있도록 지원하는 기능이 집합이다.
집합 기능은 일반적으로 프로젝트의 최상위 POM 설정 파일에서 <modules> 태그로 설정한다.

의존관계(dependency)

각 모듈 간 의존관계를 설정하는 방법은 다른 라이브러리에 대한 의존관계 설정과 같다.
하나의 프로젝트를 여러 모듈로 분리할 때 각 모듈 사이에 의존관계가 발생한다.
모듈 간에 서로 필요한 코드나 라이브러리를 공유할 수 있다.
각 모듈이 독립적으로 개발되어도 문제없이 함께 작동할 수 있도록 보장한다.



4.메이븐의 의존 라이브러리

■ 의존 라이브러리의 의존성 전이

프로젝트가 추가될 때마다 의존하는 라이브러리 수가 급격하게 증가할 수 있다.
증가한 라이브러리에 의해서 라이브러리에 대한 관리가 복잡해진다.
라이브러리의 복잡성을 극복하기 위해 메이븐은 2.0 버전부터 의존성 전이 기능을 제공한다.
의존성 전이는 한 프로젝트가 다른 프로젝트를 의존하고 있을 때 의존 대상이 다시 다른 프로젝트를 의존하고 있다면 첫 번째 프로젝트는 자동으로 의존의 의존에도 의존하게 되는 원리를 의미한다.
프로젝트 A가 프로젝트 B를 의존하고 있고 프로젝트 B가 프로젝트 C를 의존하고 있다면 프로젝트 A는 프로젝트 B를 통해 프로젝트 C에 대한 의존성을 갖게 된다.

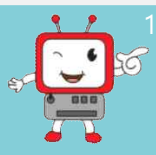
프로젝트는 자신이 직접적으로 의존하고 있는 라이브러리뿐만 아니라 라이브러리들이 필요로 하는 다른 라이브러리들까지도 사용할 수 있게 된다.
라이브러리 관리의 복잡성이 크게 줄어들게 된다.
개발자는 직접적으로 사용하는 라이브러리에만 집중할 수 있게 되고 나머지 의존성 관리는 메이븐에 맡길 수 있게 된다.
의존성 전이는 때때로 예상치 못한 버전의 라이브러리가 프로젝트에 포함되는 문제를 일으킬 수도 있다.
의존성 관리라는 메커니즘을 통해 프로젝트에서 사용할 특정 버전의 라이브러리를 명시적으로 지정할 수 있다.

✔ 의존성 중개(Dependency Mediation)

프로젝트가 같은 라이브러리의 다양한 버전에 의존할 때 발생하는 문제를 해결한다.
의존성 중개를 사용하여 라이브러리의 적절한 버전을 결정하게 된다.
라이브러리의 버전은 의존성 트리에서 가장 가까운 정의를 따르게 된다.
가까운 의미는 의존성 트리의 루트에서 해당 라이브러리까지의 거리를 의미한다.
버전이 다른 두 개의 라이브러리가 의존관계에 있다면 메이븐은 더 가까운 의존관계에 있는 버전과 의존관계를 맺는다.
A 프로젝트에서 A→B→C→D2.0 버전의 의존관계와 A→E→D1.0 버전의 의존관계가 발생한다면 A 프로젝트는 D1.0 버전과 의존관계를 맺는다.
D2.0 버전을 사용하고 싶다면 A 프로젝트의 메이븐 설정 파일에 명확하게 의존관계를 명시한다.
다양한 다른 버전의 같은 라이브러리 간의 충돌을 해결하고 프로젝트의 안정성을 유지한다.

✔ 의존성 관리(Dependency Management)

프로젝트의 모든 의존성에 대한 중앙 위치에서의 관리할 수 있게 된다.
보통 상위 레벨의 POM 설정 파일에서 의존성 관리 섹션을 정의한다.
프로젝트 전체에서 사용할 라이브러리의 버전을 명시하게 된다.
모든 모듈이 같은 라이브러리와 버전을 사용하도록 보장할 수 있다.
의존성의 버전을 변경하려면 중앙 의존성 관리 섹션만 수정하면 된다.
각각의 모듈에서 의존성의 버전을 일일이 관리하는 것을 방지한다.
전체 프로젝트의 일관성을 유지할 수 있다.
<dependencyManagement> 태그에 의존관계 라이브러리와 버전을 명시적으로 정의한다.
A 프로젝트에서 D2.0 버전을 사용한다고 <dependencyManagement> 태그에 설정한다.



4.메이븐의 의존 라이브러리

의존성 차단(Excluded Dependencies)

특정 의존성이 가져오는 전이적 의존성 중 일부를 제외하고자 할 때 사용한다.

A 라이브러리가 B 라이브러리에 의존하고 B 라이브러리는 다시 C 라이브러리에 의존하는 경우가 있다.

A 라이브러리를 사용하면 B와 C 라이브러리도 함께 가져와진다.

C 라이브러리가 필요하지 않는다면 A 라이브러리의 의존성 설정에서 C 라이브러리를 차단할 수 있다.

예 Log4j 라이브러리를 사용하기 위해서 commons-logging 라이브러리를 차단한다.

의존관계에 있는 라이브러리를 <exclusion> 태그를 활용하여 명시적으로 차단하거나 제외할 수 있다.

의존성 선택(Optional Dependencies)

특정 라이브러리가 다른 라이브러리에 의존할 때 의존성이 항상 필요하지 않을 때 사용된다.

특정 라이브러리 A가 다른 라이브러리 B를 사용하도록 설계되었지만, B가 없어도 A가 작동하는 경우가 있다.

라이브러리 B가 없어도 라이브러리 A가 작동할 때 B를 선택적 의존성으로 표시할 수 있다.

프로젝트가 라이브러리 A에 의존할 때 라이브러리 B를 가져오지 않을 수 있다.

A→B→C와 같은 구조로 의존관계를 맺는 경우 B 프로젝트에서 C가 선택적으로 설정되어 있다면 A 프로젝트를 빌드 할 때 C는 의존관계를 맺지 않도록 설정하는 기능으로 <optional> 태그를 활용하여 설정할 수 있다.

최종적으로 의존관계를 맺는 라이브러리는 소수만 남는데 그 이유는 라이브러리의 의존성 스코프와 의존성 선택 때문이다.

의존성 스코프(Dependency Scope)

현재 빌드 상태에 맞는 라이브러리일 때 의존관계를 맺는다.

단순히 라이브러리에 대한 의존성 스코프만을 정의하는 것이 아니라 의존성 전이와 관련이 있다.

다음의 표는 의존성 스코프와 의존성 전이의 관계이다.

직접적인 의존관계	의존성 전이에 있는 라이브러리의 스코프			
라이브러리의 스코프	compile	provided	runtime	test
compile	compile		compile	
provided	provided	provided	provided	
runtime	runtime		runtime	
test	test		test	

■ compile 스코프

기본 스코프로 아무것도 설정되지 않았을 때 적용된다.

■ provided 스코프

실행 시 의존관계를 제공하는 JDK나 컨테이너에서 적용된다.

프로젝트가 직접 포함할 필요가 없는 경우에 사용한다.

예 웹 애플리케이션의 생성 시 웹 컨테이너가 서블릿 API를 제공되므로 직접 포함할 필요가 없다.

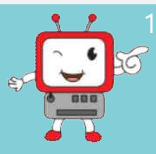
■ runtime 스코프

의존관계가 컴파일 시에는 적용되지 않지만, 실행할 시에는 적용된다.

■ test 스코프

애플리케이션 사용에 대해서는 의존관계가 필요 없고 테스트 컴파일과 실행 시점에만 적용된다.

최종 배포 산출물을 포함하는 시점에는 포함되지 않는다.

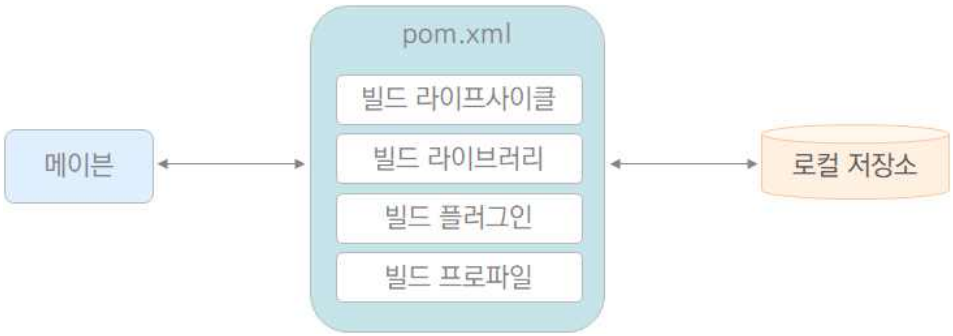


4.메이븐의 의존 라이브러리

의존 라이브러리의 POM

POM의 소개

프로젝트 객체 모델을 POM(Project Object Model)이라고 한다.
프로젝트를 구현할 때 필요한 정보를 관리할 수 있도록 정의한 모델이다.
메이븐은 빌드와 관련된 정보를 POM 설정 파일로 정의한다.
프로젝트 디렉터리 구조와 메이븐에서 사용할 수 있는 플러그인 정보를 포함하고 있는 POM 설정 파일을 제공한다.
일반적으로 프로젝트 수행 시 개발자들은 의존관계 라이브러리 설정을 위해 프로젝트 내부 디렉터리에 직접 라이브러리를 받아와 저장시키거나 외부 경로 설정을 통해 사용한다.
직접 프로젝트 내부에서 사용하는 의존관계 라이브러리 종류와 버전을 관리하면 문제가 발생할 수 있다.
POM 설정 파일인 pom.xml 파일에서 편리하게 의존관계 라이브러리를 관리할 수 있다.
POM 설정 파일은 의존관계 라이브러리를 효과적으로 처리하기 위해 의존성 관리 메커니즘을 제공하고 있다.
메이븐에서 명령을 실행하면 기본적으로 POM 설정 파일인 pom.xml 파일을 읽어 빌드를 실행한다.



POM의 저장소

중앙 저장소

메이븐의 중앙 저장소에 등록된 POM 정보는 다음의 사이트에서 검색할 수 있다.

https://search.maven.org/

원격 저장소

원격 저장소나 중앙 저장소에 등록된 POM 정보는 다음의 사이트에서 검색할 수 있다.

https://mvnrepository.com/

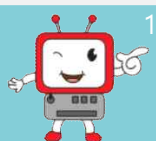
가장 많이 사용하고 있는 사이트로 검색하기가 편한 구조로 이루어져 있다.
사이트에 표시된 usages의 수는 의존 라이브러리를 사용하고 있는 라이브러리의 개수이다.

Maven 아이콘에 표시된 내용을 pom.xml의 <dependency> 태그에 복사하여 등록하고 해당 라이브러리를 사용한다.
원격 저장소에 등록되었을 때 Note에 적혀 있는 저장소의 경로를 아래와 같이 확인한다.

예 원격 저장소 경로

Note: this artifact it located at Jahia repository (http://maven.jahia.org/maven2/)

확인한 경로를 <repository> 태그에서 원격 저장소를 설정한다.
원격 저장소를 설정한 후 <dependency> 태그에서 등록하여 라이브러리를 다운로드한다.



4.메이븐의 의존 라이브러리

POM의 기능

POM 설정 파일은 프로젝트에 대한 모든 필요 정보를 포함하고 있는 XML 파일이다.
프로젝트에 대한 기본 정보와 의존성 정보, 빌드 프로세스 정보 등이 포함된다.
메이븐 빌드 도구는 POM 설정 파일을 참조하여 프로젝트를 빌드한다.
빌드 과정에서 소스 코드는 컴파일 단계에서 바이트 코드로 변환된다.
POM 설정 파일에서 설정된 컴파일러 플러그인과 소스 코드 위치 등에 의해 제어된다.
POM 설정 파일을 정의함으로써 프로젝트의 다양한 기능을 활용할 수 있다.

프로젝트 메타데이터 정의

프로젝트 이름, 버전, 작성자 등의 메타데이터를 정의한다.

의존성 관리

프로젝트가 의존하는 라이브러리들과 해당 버전을 명시하고 자동으로 다운로드하여 관리한다.
프로젝트를 위한 유일한 메이븐 코디네이터(coordinate)인 groupId, artifactId, version을 정의한다.
프로젝트가 의존성을 정확하게 관리하고 필요한 라이브러리를 효과적으로 다운로드하고 사용할 수 있게 한다.

빌드 설정

컴파일러 버전, 소스 코드 디렉터리, 테스트 환경 설정, 결과물 패키징 방법 등이 포함될 수 있다.

플러그인 관리

어떤 플러그인을 어느 단계에서 사용할 것인지 설정한다.

프로젝트 관계 관리

프로젝트 간의 상속 관계나 의존관계를 정의하여 모듈화된 프로젝트의 관리를 돕는다.
프로젝트에 의존성이 있는 다른 프로젝트의 속성으로 기술할 수 있다.

디펜던시

프로젝트가 필요로 하는 의존성을 정의하고 관리한다.
자동으로 필요한 라이브러리를 다운로드하고 적절한 위치에 배치하는 것을 포함한다.

원격 저장소

원격 저장소에서 라이브러리를 가져오는 방법을 설정한다.
중앙 저장소뿐만 아니라 사용자가 정의한 저장소도 포함될 수 있다.

통합 개발 도구에 대한 이식성과 통합

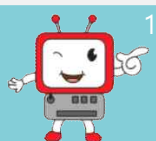
다양한 통합 개발 환경(IDE)와 호환될 수 있도록 돕는다.

프로젝트 산출물의 쉬운 검색과 필터링

메타데이터를 기반으로 산출물을 쉽게 찾고 필터링할 수 있게 돕는다.

빌드 로직의 전역적인 재사용

특정 프로젝트나 모듈에 공통으로 적용되는 빌드 로직을 재사용하기 위한 수단을 제공한다.
코드 중복을 줄이고 일관성을 유지하는 데 도움이 된다.
플러그인의 재설정이나 커스터마이징(customization)은 POM 설정 파일 안에서만 이루어진다.



4.메이븐의 의존 라이브러리

POM의 카테고리

프로젝트 기본 정보(General Project Information)

프로젝트의 기본적인 정보를 관리한다.

POM 파일에서 프로젝트에 대한 고유한 정보를 제공한다.

STS에서는 Overview 탭에서 확인할 수 있다.

코디네이터

POM.xml 파일 안에 코디네이터를 명시한다.

프로젝트를 고유하게 식별한다.

빌드 시에 생성되는 아티팩트와 아티팩트가 어떤 프로젝트에서 생성되었는지 추적할 수 있다.

라이브러리 의존성 등을 관리한다.

원격 저장소에서 필요한 라이브러리를 찾아 다운로드하는 데 사용된다.

- groupId

  프로젝트를 만든 조직이나 그룹을 고유하게 식별한다.

  일반적으로 도메인 이름을 반대로 적은 형태를 사용한다.

  예 min.spring

- artifactId

  프로젝트의 이름을 나타낸다.

  이름은 해당 그룹 내에서 고유해야 한다.

  예 test

- version

  프로젝트의 현재 버전을 나타낸다.

  프로젝트 버전의 구조는 [메이저 버전-마이너 버전-순차 버전-식별자]로 이루어진다.

  BUILD와 SNAPSHOT을 접미사로 사용할 수 있으며 SNAPSHOT 접미사는 개발 중인 버전을 나타낸다.

  SNAPSHOT 접미사를 사용하면 원격 저장소에서 최신 상태를 확인하고 필요하다면 새로 다운로드한다.

  연속적인 통합 환경에서 유용하게 사용될 수 있다.

  예 1.0.0-BUILD-SNAPSHOT

- packaging

  패키징 타입을 정의한다.

  패키징 타입에는 war, jar, ejb, ear, pom, maven-plugin 등이 있으며 war는 웹 프로젝트를 의미한다.

프로젝트 이름

프로젝트의 전체 이름을 제공한다.

사람이 읽을 수 있는 이름이며 프로젝트의 웹 사이트나 문서에 사용될 수 있다.

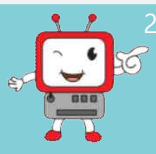
프로젝트 URL

프로젝트 웹 사이트의 URL이다.

프로젝트 설명

프로젝트에 대한 짧은 설명을 제공한다.

프로젝트 웹 사이트나 문서에 사용될 수 있다.



4.메이븐의 의존 라이브러리

프로젝트 관계 설정(POM Relationships)

프로젝트 간의 관계를 관리하며 프로젝트 대부분은 다른 라이브러리와 의존관계를 통해서 관계를 맺는다.
특별한 경우에는 각 프로젝트 간에는 상속 관계를 통해서 관계를 맺기도 한다.

의존성(Dependencies)

프로젝트가 다른 라이브러리나 프로젝트에 의존하는 관계를 설정한다.
필요한 의존성을 자동으로 다운로드하고 빌드 프로세스에 포함한다.

모듈(Modules)

하나의 프로젝트가 다수의 하위 프로젝트나 모듈을 가질 때 설정한다.
모듈을 함께 빌드하고 필요에 따라 각 모듈 간의 의존성을 관리한다.

빌드 설정(Build Settings)

메이븐 빌드와 관련한 기본적인 빌드 설정을 변경하며 기본 코드나 리소스에 대한 변경이 필요할 때 사용한다.
기본으로 제공하는 플러그인의 설정을 변경하거나 새로운 플러그인을 추가한다.
메이븐 프로젝트에서는 POM 설정 파일 내에 빌드 설정이 포함된다.

플러그인 설정

빌드 과정에서 사용될 메이븐 플러그인들과 플러그인들에 대한 설정을 포함한다.
컴파일러 플러그인, 테스트 플러그인, 패키징 플러그인 등이 있다.

빌드 프로파일 설정

다양한 빌드 환경을 위한 프로파일 설정을 포함하며 각 프로파일은 서로 다른 환경에서 빌드를 실행하기 위한 설정을 담고 있다.
개발 환경, 테스트 환경, 프로덕션 환경 등에 따라 서로 다른 프로파일을 설정할 수 있다.

라이프사이클

빌드 생명주기를 통해 빌드 과정을 정의하며 각 생명주기는 여러 단계로 구성된다.

프로젝트의 구조

프로젝트의 소스 코드 위치, 빌드 출력물의 위치, 빌드 출력물의 형식 등을 설정할 수 있다.

빌드 환경(Build Environment)

다양한 환경에 따라 다른 설정 정보를 관리하며 소프트웨어를 빌드하는데 필요한 환경을 의미한다.
모든 프로젝트는 다양한 환경에 배포하는 것이 가능해야 하며 각 환경에 따라 변경되는 부분이 발생할 수 있다.
메이븐에서는 다양한 환경에 대한 지원을 위하여 프로파일 기능을 제공한다.
메이븐의 settings.xml 파일은 빌드 환경을 설정하는 데 사용된다.
로컬 리포지토리의 위치, 원격 리포지토리, 프락시 설정 등이 포함될 수 있다.
다양한 빌드 환경에 적응하고 효율적인 빌드를 수행할 수 있다.

로컬 빌드 환경

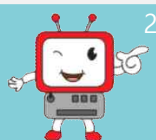
개발자의 개인 컴퓨터에서 빌드를 수행하는 환경으로 개발자의 작업환경에 따라 크게 달라질 수 있다.

통합 빌드 환경

버전 컨트롤 시스템에서 코드를 가져와서 테스트하고 빌드하는 서버 환경이다.
소프트웨어의 품질을 지속해 확인하고 문제를 빠르게 발견할 수 있다.

프로덕션 빌드 환경

실제 사용자에게 배포될 소프트웨어를 빌드하는 환경으로 보안, 성능, 안정성 등이 중요하게 다루어진다.



4.메이븐의 의존 라이브러리

POM의 설정

프로젝트 선언

프로젝트의 설정

형식

```
<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 http://maven.apache.org/maven-v4_0_0.xsd">
  ... 빌드 설정 ...
</project>
```

프로젝트를 선언하며 빌드를 설정한다.

프로젝트의 태그

프로젝트 설정

시작과 끝 태그로 선언한다.

최상위 태그다.

모든 빌드를 설정할 때 사용한다.

- <project> 태그

  모든 빌드를 설정한다.

  - xmlns 속성 : 접두사를 명시하지 않고 네임스페이스를 선언하기 위해서 사용한다.
  - xmlns:xsi 속성 : XML 스키마 인스턴스 네임스페이스에 속한 속성을 사용할 수 있다.
  - xsi:schemaLocation 속성 : 스키마 위치를 지정하며 XML 문서에서 사용되는 스키마의 위치를 URI 형식으로 설정한다.

디펜던시 선언

디펜던시의 설정

형식

```
<dependencies>
  ... 의존 라이브러리 설정 ...
</dependencies>
```

개발자가 의존관계가 있는 라이브러리를 등록하여 실질적인 관리를 한다.

모든 의존관계가 있는 라이브러리를 하위 태그인 <dependency> 태그로 등록한다.

라이브러리 개수인 <dependency> 태그의 개수에 제한이 없다.

의존 라이브러리를 제공하는 정보에 의해서 작성한다.

디펜던시의 태그

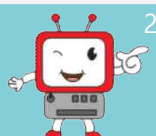
디펜던시 관리

시작과 끝 태그로 선언한다.

프로젝트에 의존관계가 있는 라이브러리를 관리할 때 사용한다.

- <dependencies> 태그

  프로젝트에 의존관계가 있는 라이브러리를 관리한다.



4.메이븐의 의존 라이브러리

POM의 태그

프로젝트 기본 설정

프로젝트의 기본 설정에 관한 내용을 선언한다.
프로젝트 설정 내용은 Overview 탭에서도 확인할 수 있다.

버전 관리

- <modelVersion> 태그

  POM 모델의 버전을 관리하며 시작과 끝 태그로 선언한다.

고유 아이디 설정

- <groupId> 태그

  생성하는 프로젝트의 고유 아이디를 설정하며 시작과 끝 태그로 선언한다.
  고유 아이디는 일반적으로 도메인 이름을 사용한다.

유일 아이디 설정

- <artifactId> 태그

  프로젝트를 식별하는 유일한 아이디를 설정하며 시작과 끝 태그로 선언한다.

프로젝트 이름

- <name> 태그

  프로젝트의 이름을 설정하며 시작과 끝 태그로 선언한다.

프로젝트 패키징

- <packaging> 태그

  프로젝트의 패키징을 설정하며 시작과 끝 태그로 선언한다.

프로젝트 버전

프로젝트 개발 중에는 BUILD와 SNAPSHOT을 접미사로 사용할 수 있다.
SNAPSHOT을 접미사로 사용할 수 있으며 SNAPSHOT 접미사는 개발 중인 버전을 나타낸다.
SNAPSHOT 접미사의 기능은 다음과 같다.

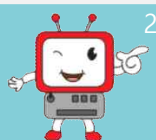
- 빌드 할 때마다 가장 최근에 배포한 라이브러리가 있는지를 파악한다.
- 로컬 저장소에 존재하는 버전보다 최신 버전의 라이브러리가 있을 때 다운로드한다.
- 변경되는 코드에 대하여 지속해서 배포하고 참조하는 것이 가능하다.
- 현재 버전을 SNAPSHOT 버전으로 유지하여 프로젝트를 완료한다.
- 배포하는 시점에 최종 버전으로 관리한다.

- <version> 태그

  프로젝트의 현재 버전을 설정하며 시작과 끝 태그로 선언한다.

예시

```
<modelVersion>4.0.0</modelVersion>
<groupId>com.example</groupId>
<artifactId>Servlet</artifactId>
<version>1.0-SNAPSHOT</version>
<name>Servlet</name>
<packaging>war</packaging>
```



4.메이븐의 의존 라이브러리

프로퍼티 설정

프로퍼티 설정

- <properties> 태그 : 프로퍼티를 설정하며 시작과 끝 태그로 선언한다.

소스 인코딩 설정

- <project.build.sourceEncoding> 태그 : Maven 프로젝트의 소스 코드 인코딩을 설정하며 시작과 끝 태그로 선언한다.

컴파일 타겟 JDK 버전 설정

- <maven.compiler.target> 태그 : 컴파일된 클래스 파일이 호환될 JDK 버전을 지정하며 시작과 끝 태그로 선언한다.

컴파일 소스 JDK 버전 설정

- <maven.compiler.source> 태그 : 소스 코드를 컴파일할 때 사용할 JDK 버전을 설정하며 시작과 끝 태그로 선언한다.

예시

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.target>17</maven.compiler.target>
  <maven.compiler.source>17</maven.compiler.source>
  <junit.version>5.10.2</junit.version>
</properties>
```

의존 라이브러리의 등록

라이브러리 등록

- <dependency> 태그 : 의존관계가 있는 라이브러리를 등록하여 다운로드하며 시작과 끝 태그로 선언한다.

고유 아이디 설정

- <groupId> 태그 : 라이브러리의 고유 아이디를 설정하며 시작과 끝 태그로 선언한다.

유일 아이디 설정

- <artifactId> 태그 : 라이브러리를 식별하는 유일한 아이디를 설정하며 시작과 끝 태그로 선언한다.

라이브러리 버전 설정

- <version> 태그 : 라이브러리의 버전을 설정하며 시작과 끝 태그로 선언한다.

라이브러리 범위 설정

- <scope> 태그 : 라이브러리의 적용 범위를 설정하며 시작과 끝 태그로 선언한다.

예시

```
<dependency>
  <groupId>jakarta.servlet</groupId>
  <artifactId>jakarta.servlet-api</artifactId>
  <version>6.0.0</version>
  <scope>provided</scope>
</dependency>
```