


DZone (/) > Java Zone (/java-jdk-development-tutorials-tools-news) > Dependency Management and Versioning With a Maven Multi-Module Project

Dependency Management and Versioning With a Maven Multi-Module Project



(/users/2897503/avraam.html) by **Avraam Piperidis** (/users/2897503/avraam.html)

 MVB </> CORE ·

Jul. 11, 19 · Java Zone (/java-jdk-development-tutorials-tools-news) · Tutorial

 Like (14)  Comment (1)  Save  Tweet



O'Reilly | Build and Deploy Serverless Applications with Java

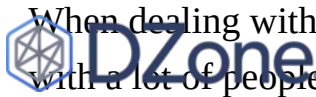
Learn to build & deploy serverless Java apps with AWS Lambda. Completely free: All 10 chapters of O'Reilly's Programming

In this article, we are going to look at how to implement a multi-module project in Maven with versioning and dependency management, as well as the best practices for building big, large-scale projects from both a developer perspective and a DevOps/management perspective.

However, if you are not familiar with Maven, I highly recommend reading this (<https://maven.apache.org/guides/getting-started/index.html>) article first and getting some experience using Maven. This article will not cover Maven basics.

So, with that in mind, let's get started.

Introduction

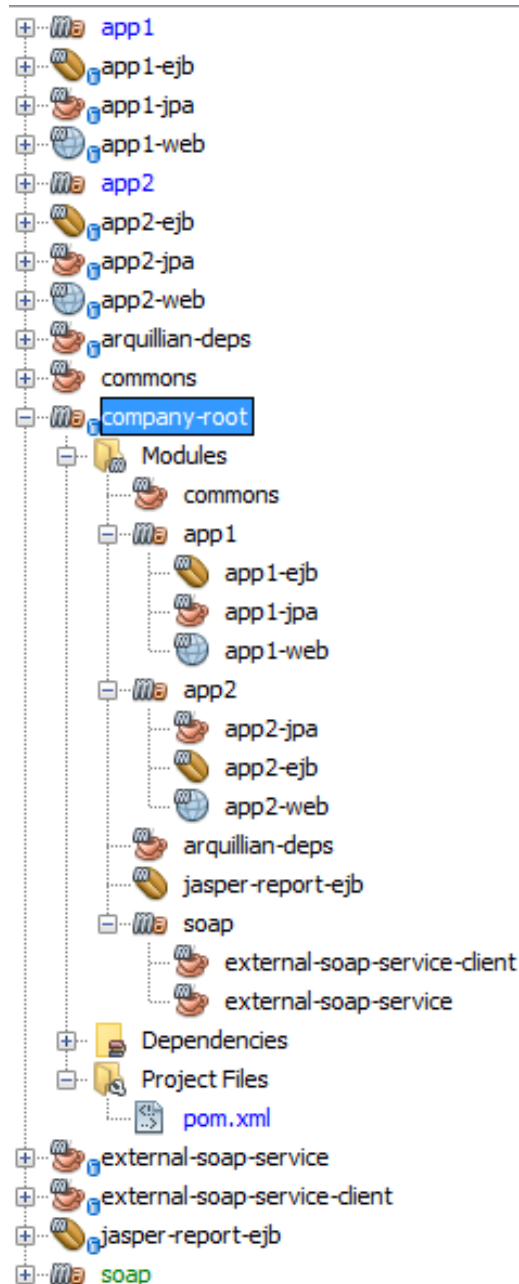


When dealing with large-scale software, we have to communicate, work, and cooperate with a lot of people — whether in an organization or a community. Besides that, we have to deal with the environment we are working with, which may consist of many projects, large or small, external projects, libraries, shared modules, utilities, and many others.

Below, we are going to see how Maven can make our lives a lot easier.

Master-Root POM Project

This is the desired structure we want to accomplish:



I used NetBeans 8.2, but it can be accomplished with simple mvn commands.

Now, we begin by creating a POM project company-root similar in the picture. The

following are the important parts of the pom.xml:

[\(/users/login.html\)](/users/login.html)[\(/search\)](/search)

The `packaging` tag defines the type of project, which is a type of POM:

[REFCARDZ \(/refcardz\)](#) [TREND REPORTS \(/trendreports\)](#) [WEBINARS \(/webinars\)](#) [ZONES](#) ▾

```
1 <packaging>pom</packaging>
```

A POM is basically a container of sub-modules.

The `properties` tag:

```
1 <properties>
2   <appx.version>1.0.0</appx.version>
3 </properties>
```

`Properties` are value placeholders. Their values are accessible anywhere within POM using the notation `${X}` where X is the property.

We are going to define versioning in the `properties` but will explain later.

The `dependencyManagement` tag:

```
1 <dependencyManagement>
2   <dependencies>
3     ...
4   </dependencies>
5 </dependencyManagement>
```

A very important use of the dependency management section is to control, consolidate, and centralize the versions of artifacts used in dependencies and inherited by all children.

So, what we are going to do is define the company-root pom.xml as a base/parent project, define the dependencies that are going to be used, and set the versioning and the child projects.

A Simple Real-Life Scenario

We are going to define the usage of the JUnit framework, version, and scope for the projects below company-root. This is really useful when having dozens or more projects because you know exactly what version of dependencies they have. The projects don't define the versions themselves, rather the people who control the company root project

```
1 <properties>
2   <junit.version>4.12</junit.version>
3 </properties>
```

And then, the dependency:

```
1 <dependencyManagement>
2   <dependencies>
3     <dependency>
4       <groupId>junit</groupId>
5       <artifactId>junit</artifactId>
6       <version>${junit.version}</version>
7       <scope>test</scope>
8     </dependency>
9   </dependencies>
10 </dependencyManagement>
```

With the above declaration, everyone will be using JUnit 4.12 with a test scope.

The declaration of JUnit on child projects is as simple as that.

```
1 <dependency>
2   <groupId>junit</groupId>
3   <artifactId>junit</artifactId>
4 </dependency>
```

Now, let's assume the DevOps engineer wants to test everything with the new JUnit 5 Framework. The only action required for him would be to change the property version on company-root pom from 4.12 to 5.0 and run the tests.

More info on the Maven dependency mechanism can be found here



Also, let's take a moment for a quick recap:

- Bug fix: just bug fixes and related stuff.
- Minor: Improvements, new features, deprecation notices, don't break user code (backward-compatibility). Same API.
- Major: new features, to complete API changes.

More info for software versioning can be found here (<https://semver.org/>)

A Common Project

In the next step, we want to create a base framework with core functionality or a library with common utilities for all of our projects to use, extend, or explore whatever the usage is.

To succeed that, we created a new Maven Java application with a name called common. If we open pom.xml of the newly created project, we observe that the parent section is missing.

Let's adjust it by concluding a parent section so commons will turn into a child (or a Leaf POM, a child with packaging other than POM) of the company-root project.

After we finish editing pom.xml, the outcome is as follows:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/200
3     <modelVersion>4.0.0</modelVersion>
4     <groupId>com.protectsoft.company</groupId>
5     <artifactId>commons</artifactId>
6     <packaging>jar</packaging>
7     <version>1.0-SNAPSHOT</version>
8
9     <parent>
```



```

10 <groupId>com.protectsoft</groupId>
11 <artifactId>company</artifactId>
12 <version>1.0-SNAPSHOT</version>

```



(/users/login.html)



(/search)

REGARDZ (/regardz)

TREND REPORTS (/trendreports)

WEBINARS (/webinars)

ZONES ▾

```

20 <dependencies>
21 <dependency>
22 <groupId>junit</groupId>
23 <artifactId>junit</artifactId>
24 </dependency>
25 </dependencies>
26 </project>

```

If you are not familiar with Maven, I highly recommend reading this (<https://maven.apache.org/guides/getting-started/index.html>) article first and establishing some experience around Maven.

Two important things we need to mention:

- 1) The parent section. Now, this project has an inheritance.
- 2) The dependency section. We defined a JUnit dependency without version or scope because they are inherited. That means the usage of the JUnit framework is predefined from someone else.

Now, let's update company-root POM and add a version for commons project for others to use and keep things organized. To do this, we add the following in the company-root POM:

```

1 <properties>
2 <!-- Our projects versioning -->
3 <company.commons>1.0-SNAPSHOT</company.commons>
4
5 <!-- External dependencies versioning -->
6 <junit.version>4.12</junit.version>
7 </properties>
8
9 <dependencyManagement>
10 <dependencies>
11 <dependency>
12 <groupId>com.protectsoft.company</groupId>
13 <artifactId>commons</artifactId>
14 <version>${company.commons}</version>

```



```
14 <version>${company.common.version}</version>
15 </dependency>
16 <dependency>
17 <groupId>junit</groupId>
18 <artifactId>junit</artifactId>
19 <version>${junit.version}</version>
```



(/users/login.html)



(/search)

REGARDZ (/refcardz)

TREND REPORTS (/trendreports)

WEBINARS (/webinars)

ZONES ▾

Creating a 3-Tier Web Application

Here, we are going to create our first web application using 3-tier architecture. But we want this application to be developed into three separate sub-projects and glued all together as one.

We are considering this stack for the 3-tier.

1) Presentation Layer

REST

2) Bussines/LogicLayer

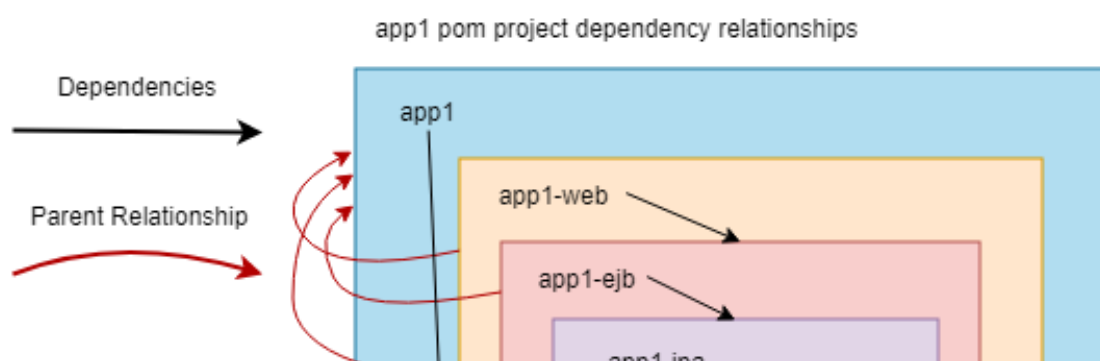
EJB Enterprise Java Beans

3) Data Access Layer

JPA Java Persistence Object

First, in the presentation layer, we have the endpoints to be consumed by clients through HTTP. We might say it is an API for clients. In the business layer, this is where you can find all of the logic and functionality that really matters, and finally, you can also find the data access layer, which is the store and access of data through persistent storage.

The desired result we want to achieve is this:





exposes the endpoints. App1-web is the presentation layer of our 3-tier. App1-web project contains a dependency of app1-ejb project, which is the business layer. The app1-ejb contains a dependency of app1-jpa, which is the data access layer and mostly contains entity classes and metadata information. The parent project from app1-web, app1-ejb, and app1-jpa is the app1.

After we create the app1 POM project, the important sections of pom.xml is the following:

```

1 <packaging>pom</packaging>
2 <parent>
3   <groupId>com.protectsoft</groupId>
4   <artifactId>company</artifactId>
5   <version>1.0-SNAPSHOT</version>
6 </parent>
7
8 <dependencies>
9   <dependency>
10    <groupId>com.protectsoft.company</groupId>
11    <artifactId>commons</artifactId>
12  </dependency>
13  <dependency>
14    <groupId>junit</groupId>
15    <artifactId>junit</artifactId>
16  </dependency>
17 </dependencies>

```

This project has company-root as its parent, and in dependencies, the version is inherited.

Now, let's create the three separate sub-projects. We can do it in two different ways. One is from the NetBeans IDE, which will update parent POM with sub-modules section and add a parent to the children, and the second is by hand and we have to edit pom.xml in the parent and children.

We are going to do it from the NetBeans IDE. Eclipse or IntelliJ have the same support.



We do that three times for `app1-jpa` , `app1-ejb` , and `app1-web` .

After that, we going to edit the `pom.xml` for the new three projects.

We open `app1-jpa` POM and we see that it has `app1` as a parent. This is also present in `app1-ejb` and `app1-web` . All three projects have `app1` as a parent.

```
1 <parent>
2   <groupId>com.protectsoft.company</groupId>
3   <artifactId>app1</artifactId>
4   <version>1.0-SNAPSHOT</version>
5 </parent>
```

In `app1-ejb` , we add the `app1-jpa` as a dependency:

```
1 <dependencies>
2   <dependency>
3     <groupId>${project.groupId}</groupId>
4     <artifactId>app1-jpa</artifactId>
5   </dependency>
6 </dependencies>
```

And in `app1-web` , the `app1-ejb` is a dependency.

```
1 <dependencies>
2   <dependency>
3     <groupId>${project.groupId}</groupId>
4     <artifactId>app1-ejb</artifactId>
5   </dependency>
6 </dependencies>
```

We have completed the relationship in the picture above.

Next, we going to define versioning and dependency management in the `app1 pom.xml`



1 <!-- app1 pom will define what version of child projects/modules can be used -->
 2 <properties>

```

8 <dependencies>
9   <dependency>
10     <groupId>${project.groupId}</groupId>
11     <artifactId>app1-ejb</artifactId>
12     <version>${app1.ejb.version}</version>
13   </dependency>
14   <dependency>
15     <groupId>${project.groupId}</groupId>
16     <artifactId>app1-jpa</artifactId>
17     <version>${app1.jpa.version}</version>
18   </dependency>
19 </dependencies>
20 </dependencyManagement>
21
22 <!-- app1 has this dependencies to be used be the sub-modules -->
23 <dependencies>
24   <dependency>
25     <groupId>com.protectsoft.company</groupId>
26     <artifactId>commons</artifactId>
27   </dependency>
28   <dependency>
29     <groupId>junit</groupId>
30     <artifactId>junit</artifactId>
31   </dependency>
32 </dependencies>
33

```

Reactor

We also notice this new section in the `app1 pom.xml`

```

1 <modules>
2   <module>app1-ejb</module>
3   <module>app1-jpa</module>
4   <module>app1-web</module>
5 </modules>

```

Meaning that the `app1 POM` project also has the role of the aggregator. That means the



DZone®

app1 project will build all sub-modules/projects defined in the modules section with a specific order that is analyzed by the Reactor.

[REFCARDZ \(/refcardz\)](#)
[TREND REPORTS \(/trendreports\)](#)
[WEBINARS \(/webinars\)](#)
[ZONES ▾](#)
<https://maven.apache.org/guides/mini/guide-multiple-modules.html>

```
scanning for projects...
```

```
-----
Reactor Build Order:
```

```

app1
app1-jpa
app1-ejb
app1-web
```

That means that the reactor analyzed all the projects in modules and their dependencies and a specific build order is generated. Also, cyclic dependencies are not allowed. In the above picture, first, we have `app1-jpa` as a result of having no dependencies; next, the `app1-ejb` can be built after `app1-jpa` because it is using it, and finally, the `app1-web` takes place after `app1-ejb` is completed.

Reactor summary prints the success output.

```
Reactor Summary:
```

```

app1 ..... SUCCESS [0.609s]
app1-jpa ..... SUCCESS [1.166s]
app1-ejb ..... SUCCESS [2.174s]
app1-web ..... SUCCESS [1.847s]
```

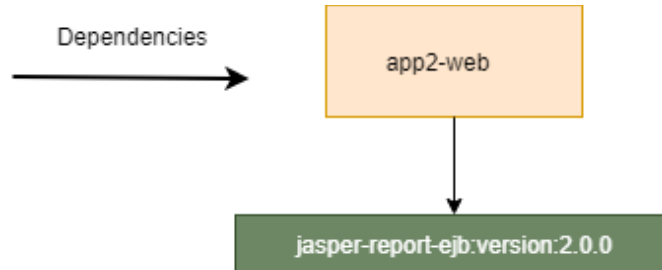
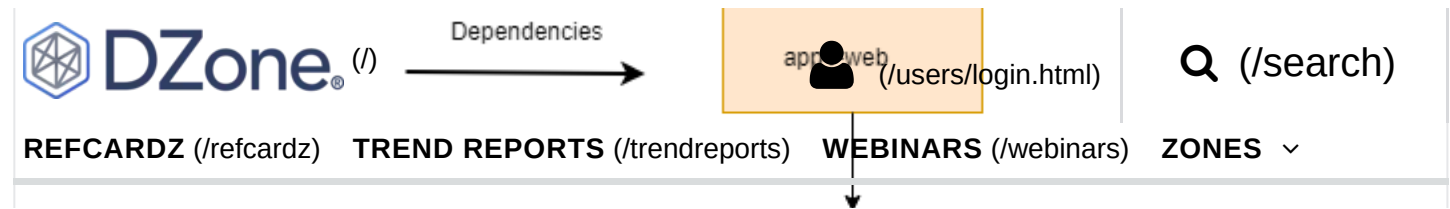
```
-----
BUILD SUCCESS
-----
```

Reactor and project aggregation are very useful in packaging and testing environments.

Using a Different Version for Different Projects

What to do in a situation where a version X of a library must be used in the project `app1` and version Y in the project `app2` ?

We are in a situation where QA management decided that the usage of a reporting library will be version 1 for `app1` and version 2 for `app2` . `App2` needs some extra features that only version 2 provides.



We cannot set the version in the company-root as one property because only one version will take place.

One solution to this new requirement is to define dependency management and versioning at the `app1` and `app2` projects.

So, `app1` POM can include this for version 1.

```

1 <properties>
2 <report.version>1.0.0</report.version>
3 </properties>
4
5 <dependencyManagement>
6   <dependencies>
7     <dependency>
8       <groupId>com.protectsoft.company</groupId>
9       <artifactId>jasper-report-ejb</artifactId>
10      <version>${report.version}</version>
11    </dependency>
12  </dependencies>
13 </dependencyManagement>

```

And `app2` POM includes the following:

```

1 <properties>
2 <report.version>2.0.0</report.version>
3 </properties>
4
5 <dependencyManagement>
6   <dependencies>
7     <dependency>
8       <groupId>com.protectsoft.company</groupId>

```



App1 and App2 are responsible and in control of the context of the sub-module projects. Furthermore, changes can take place more easily.

Last But Not Least, the Profile Section

At this point, we have to mention the profile (<https://maven.apache.org/guides/introduction/introduction-to-profiles.html>) section and its use. A lot of the stuff being said above can be grouped into different profiles. For example, we can have one profile for the **X** version of our commons library and another profile for the **Y** version of commons. Another example would be to have different profiles for the **environments** like **test** environment and a **demo** environment.

We are going to modify the parent root `pom.xml` and add two profiles. One is versioning for Java EE 7 and the other for Java EE 8.

```

1 <profiles>
2   <profile>
3     <id>java-ee-7-profile</id>
4     <properties>
5       <javax.version>7.0</javax.version>
6     </properties>
7   </profile>
8   <profile>
9     <id>java-ee-8-profile</id>
10    <properties>
11      <javax.version>8.0</javax.version>
12    </properties>
13  </profile>
14 </profiles>
```

And with Maven, we can run:

- `mvn clean install -P java-ee-7-profile` OR
- `mvn clean install -P java-ee-7-profile, java-ee-8-profile` for both profiles.



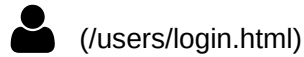
- `<dependencies>`
- `<plugins>`
- `<properties>` (not actually available in the main POM, but used behind the scenes)
- `<modules>`
- `<reporting>`
- `<dependencyManagement>`
- `<distributionManagement>`
- a subset of the `<build>` element, which consists of:
 - `<defaultGoal>`
 - `<resources>`
 - `<testResources>`
 - `<finalName>`

Conclusion

When following these steps, or setting up a similar architecture, be sure none of this is taken for granted. Rather, one should study and research what the needs and requirements really are, and take actions accordingly. This article is meant to provide a more general look at building multi-module projects with Maven, providing ideas and solutions to more general problems.

The complete git repo is available here

(https://github.com/avraampiperidis/maven_multi_module_project_example).



Opinions expressed by DZone contributors are their own.

[REPCARDZ \(/repcardz\)](#)

[TREND REPORTS \(/trendreports\)](#)

[WEBINARS \(/webinars\)](#)

[ZONES ▾](#)

fromrel=true)

- [Top 20 Git Commands With Examples \(/articles/top-20-git-commands-with-examples?fromrel=true\)](/articles/top-20-git-commands-with-examples?fromrel=true)
- [What If You Don't Want To Be a Developer Anymore? w/ CTO Connection's Peter Bell \(/articles/what-if-you-dont-want-to-be-a-developer-anymore-w?fromrel=true\)](/articles/what-if-you-dont-want-to-be-a-developer-anymore-w?fromrel=true)
- [How To Perform FastAPI Path Parameter Validation Using Path Function \(/articles/how-to-perform-fastapi-path-parameter-validation-u?fromrel=true\)](/articles/how-to-perform-fastapi-path-parameter-validation-u?fromrel=true)

Java Partner Resources

ABOUT US

[About DZone \(/pages/about\)](/pages/about)

[Send feedback \(mailto:support@dzzone.com\)](mailto:support@dzzone.com)

[Careers \(https://devada.com/careers/\)](https://devada.com/careers/)

[Sitemap \(/sitemap\)](/sitemap)

ADVERTISE

[Advertise with DZone \(https://advertise.dzzone.com\)](https://advertise.dzzone.com)

CONTRIBUTE ON DZONE

[Article Submission Guidelines \(/articles/dzzones-article-submission-guidelines\)](/articles/dzzones-article-submission-guidelines)

[MVB Program \(/pages/mvb\)](/pages/mvb)

[Become a Contributor \(/pages/contribute\)](/pages/contribute)

[Visit the Writers' Zone \(/writers-zone\)](/writers-zone)

LEGAL

[Terms of Service \(/pages/tos\)](/pages/tos)

[Privacy Policy \(/pages/privacy\)](/pages/privacy)

CONTACT US

600 Park Offices Drive

Suite 300

Durham, NC 27709

