4. Dependencies

Contents

[1. Introduction 1](#_Toc12960)

[2. Dependency Overview 1](#_Toc2519)

[3. Versions 2](#_Toc24352)

[4. Types 2](#_Toc11690)

[5. Transitive Dependencies 2](#_Toc31050)

[6. Scopes 2](#_Toc18782)

[7. Demo: Overview 2](#_Toc30261)

[8. Demo: Review of Concepts 3](#_Toc13995)

[9. Summary 3](#_Toc1481)

# 1. Introduction

=>slides: Pg. 1

In this module, we will cover all of the major aspects of dependencies and how they affect your application.

=>slides: Pg. 2

We will cover versions, types, what transitive dependencies are, and how they are useful to your application. This typically happens to be the number one reason why people start using Maven. We will also cover scopes of dependencies and what it means to have a test scope versus provided runtime and compile scopes.

# Dependency Overview

=>slides: Pg. 3

Dependencies are simply other resources that we want to use inside of our application. Maven will pull in transitive dependencies based off the dependencies we list. To use a dependency, we simply just list that dependency and Maven will automatically download it for us and pull in the transitive dependencies. To use dependencies, we have to have three things. We need to have a group ID. Group IDs are typically the same as our package structure. The next thing we need is the artifact ID. The artifact ID is these name of the actual item that we want to use. And lastly, we need a version. Version is simply the version number, nothing more, nothing less. We're going to cover versions in another slide here in a second because there's one particular thing you need to know about versions. But to add a basic dependency to our application, we just need to add XML similar to this. So, this entire dependencies section of our POM we have our dependencies in here now called org.apache.commons, that's our groupId, our artifactId is commons‑lang3, and then our version is 3.8.1. Now, this example I chose on purpose because, historically, Apache didn't follow the standard naming conventions, but they have since adapted what the industry used and dictated.

# Versions

=>slides: Pg. 4

Versions are the release number of the artifact that we want to use. There's really only one version that warrants taking time to discuss, and that is SNAPSHOT. All of your internal development should start off as a SNAPSHOT. You should also be aware of using SNAPSHOTS from other third‑party libraries. Say you're using a version of Spring or some other third‑party tool out there, and you want the latest code. You can pull it down as a SNAPSHOT. Now SNAPSHOT allows us to push new code up to a repository or a development team every time. So when we say we're using a SNAPSHOT version, and it has to be labeled specifically that in all capital letters, I'm going to show you an example of this. Every time we go to compile, it will look to see if there's new code that it should pull down and use. Besides SNAPSHOT releases, everything else of a naming convention is left up to the end user. Here's an example of an application we could create, myapp‑1.0‑SNAPSHOT. Myapp is the artifactId, as we talked about in the previous slide, and 1.0 is the version, but it's also a type of SNAPSHOT. So it's a 1.0‑SNAPSHOT, and it has to be all capital letters, as I've mentioned a few times. It does not work If it's lowercase. Our changes are always downloaded by using this convention. This is the key reason to use it though. It saves us from a re‑releasing versions of development. I've worked with a lot of organizations that have started off saying, Oh, SNAPSHOTS scare us, so we're going to release every time we make a change, and you end up having tons of releases, so your versionId ends up being something like 1.0.143. There are 143 maintenance releases. So you've got this long number instead of SNAPSHOT, which then people aren't always the latest or stuck on an older version, and it kind of defeats the purpose of it. SNAPSHOTS are good, you just have to be aware of what they are and how you should consider using them in your development lifecycle. You never want to deploy to production with a SNAPSHOT because we can't reproduce or recreate the code. The next time they go to compile it, the functionality could be different underneath for us. As we mentioned a minute ago, a release doesn't have to be a specific naming convention. Our final release could be myapp‑1.0.jar, which is very common for a naming convention. And as we do maintenance releases, that could upgrade to 1.0.1 just because we did a bug fix and didn't release any new functionality. As we release new functionality, it would maybe be 1.1. Now some industry‑common terms that don't really affect Maven, but it kind of helps you conceptualize or wrap your head around how this works are things like milestone releases. Often you'll see open source projects use 1.0‑M1 for a milestone release, or RC1, or RC2, or RC3 for release candidates. Again, not something that you should probably go to production on, but if you're tired of SNAPSHOTS possibly breaking your code, and you want people to know that you're going to release something, but it's not completely stable yet, then use a milestone or a release candidate. You'll also see some people, when they do their final release, name it as release, just because they want the convention of a SNAPSHOT where you can see whether it's a SNAPSHOT or a release, and in the final version, they'll name it ‑RELEASE. Also, some people will choose FINAL for the final release.

# Types

=>slides: Pg. 5

Types refer to the type of resource that we want to include inside of our application. The current core packaging types are pom, jar, maven‑plugin, ejb, war, ear, rar, and par. Now, jar, war, ear, rar, and par are all just glorified zip files, for all intents and purposes. The default, and most often used type, is a jar. The one here that's interesting to me, though, is the type of pom. If we do something of type pom, all of the dependencies inside that pom are downloaded into our application, and they actually refer to it as a dependency pom. The dependency pom, say you have web services in your organization and you want to group all of these dependencies that we use for any time we do a web service, you know, jersey libraries or some other libraries, and there's a bunch of different XML or JSON dependencies that we have, you can put those in a RESTful services pom and reference that pom and it will download that into our application. Packaging types refer to packaging inside of our application. If we're building an artifact for other people to consume, we are specifically going to reference that with our packaging here. The artifact that we want to use has a pom, and its pom has a packaging set on it of jar, war, ear, or pom, as I mentioned. And that's what gets pulled into our application. Your packaging and the type that somebody is going to request, what your artifact is are the same thing.

# Transitive Dependencies

=>slides: Pg. 6

Transitive dependencies, without a doubt, are the main reason people begin using Maven. If we add a dependency, something like bibernate, it will go ahead and pull down any and all transitive dependencies that Hibernate needs. Now if there are any conflicts, it will also try and resolve those too by always choosing the newer version. That doesn't always work, and we'll talk more about how to exclude dependencies here in a minute, but transitive dependencies are a great thing because, as you can see from this example I posted up here, If I put hibernate‑core, and I want to use hibernate‑core, just that artifact, inside my application, well, there's all these other JARs that Hibernate needs, antler, dom4j, the hibernate‑commons annotations,core, the jpa specifications, a bunch of logging APIs, those will automatically be downloaded into my application. Now transitive dependencies often scare or frighten people because they don't know where all of these things are coming from. When the people who created Hibernate, or whatever library you're using, sat down and programmed it, they knew what libraries it needed to work with. Who better then to decide what versions and what different libraries it should pull in for your application then the people who are creating the framework that you're using? In this case, Hibernate 5.4.1 specifically needs antler 2.7.7, dom4j 2.1.1, they know the JPA specification it's supposed to work with. The POM that you download from them actually defines those dependencies that it's looking for, and only in the case where there's a conflict will it try and override it with something newer.

# Scopes

=>slides: Pg. 7

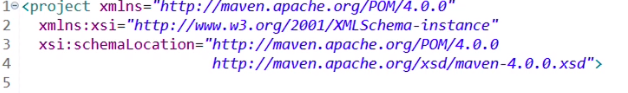
Scopes. A lot of people use Maven without ever taking advantage of scopes. You can get along fine with that if you just want to use it to download dependencies and transitive dependencies and it's great. There are some scopes though that are particularly worth mentioning that I want to point out to you. I'm going to run through what these six scopes available for your dependencies are and highlight a couple of key ones for you. Compile is the default scope. You don't have to explicitly list it, it's implicitly implied. In the previous example I had it on there. If I had omitted it, it'll compile just fine. That's what the default is. It's available. It means all of my resources are available everywhere inside of my application. The next one that's probably the main one that you should know is provided. Provided is a lot like compile. It means that the artifact is going to be available throughout the entire build cycle. In one of the previous modules we talked about phases, the different phases that we have. You know, test and when we go to package, and we have everything that all of our artifact is going to be available throughout all of the code, but that's not going to package it up at the end. A good example of this is something like the Servlet API or the XML APIs. It's available in our phases, but not included in the final artifact. And we don't want to include it because it will be provided by the container that we are deploying our app to. Another interesting one is runtime. Runtime is not needed for compilation, but it's needed for execution. So dynamically loaded libraries are things like JDBC jars, items that we need a driver that's dynamically loaded at runtime. Say we're using JDBC to connect to a database. We're going to do a Class.forName or a DriverManager load resource, something like that, something that's dynamically loaded. We need that bundled with our application. It's actually kind of the opposite of provided. You don't need it for testing, you don't even need it for compiling, you don't need to package it, but you need it for your application to run. Now another interesting scope is the test scope. Testing is only available for the test compilation and execution phase. It does nothing for compiling. It does nothing for packaging. It's also not included in our final artifact. A good example of this would be the JUnit or TestNG jars. Now system, I'm going to point out to you to never use. System is very brittle and actually breaks the whole reason of why we want to use Maven. I have seen people use this once or twice before and it's always lead to problems. Basically, you're going to hard code a path to a jar on your file system. This is why we're using Maven. Don't use system. Why do they have it, you might ask, to just simply tie existing projects into a Maven build. The last one is import. This is an advanced topic. It deals with dependency management, which is something we're not going to cover in this course. Dependency management deals with sharing resources across multiple POMs, and it's kind of an advanced topic that we're just not going to cover here.

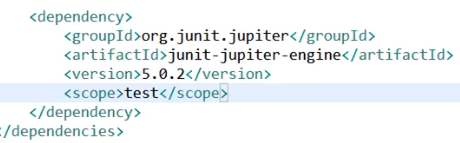
# Demo: Overview

=>slides: Pg. 8

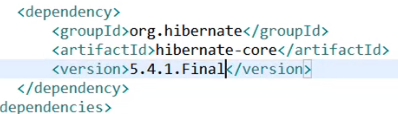
Let's demo some of the concepts that we just spoke about. In our demo, we're going to look at the pom.xml file and add a dependency to it. We'll show you how to add that dependency not used in our HelloWorld application currently, and show you how to reference to transitive dependencies that are pulled in. Then we'll also use scopes to show you how things don't get packaged or do get packaged in our final artifact.

# Demo: Review of Concepts

So while developing this, I had an error message. It was actually a warning telling me that there wasn't a schema defined inside of our application. So I went ahead and copied Maven's project schema namespace up to the top of our project. So if you look at the screen now, lines 1 through 4, I've added just so it doesn't keep complaining that there's not a schema defined inside of our app. It runs fine without it, but you should have this so that the context‑sensitive help works better for whichever tool you're using. Let's go ahead and dive into how we add some dependencies inside of our application. We already added the dependency for org.apache.commons commons‑lang3 when we were showing some of the other examples before. But I want to show you the Dependencies tab down here at the bottom of my screen. If you're inside of Spring STS, it'll show you what you have for dependencies inside of your application. Now if we switch back to the pom.xml tab and I add another dependency in here, you're going to see the Dependencies tab automatically get updated. So let's start by adding in here another dependency. And if you did not add the commons‑lang dependency before, go ahead and add it now. Make sure you have a dependencies tag like on line 12 and 18 and then a nested dependency element inside of there like on line 13 and 17. I'm going to give myself a little extra space down below here and add another dependency element inside of here.



And I want to add a <groupId> of org.junit.jupiter. This is the new API for JUnit 5 to run unit tests inside of your application. So I've got a groupId, now let's add an artifactId, and we want to put in here junit‑jupiter‑engine. And I wanted to show this example for two reasons. First, this is a POM, and it will download the other two dependencies it needs, and we want to show scoping as well. So for our version, let's do 5.0.2, and then for our scope, let's put in test and save this. Now, it'll build your workspace and we can switch over to the Dependencies tab, and you see inside of here, we have the junit‑jupiter‑engine, but if I switch to Dependency Hierarchy, you'll see it has started to pull in transitive dependencies. So commons‑lang didn't have any transitive dependencies, but the junit‑jupiter‑engine POM file pulls in version 1.0.2 of the engine and 5.0.2 of the jupiter‑api. You'll also see that it has started to resolve some conflicts. So with junit‑platform‑commons, it omitted 1.0.2 for a conflict, so it chose which version it needed to based off of what it needed to rev up to to work with other libraries. Let's switch back to the POM file, and let's add one more dependency here.



We want to add in the hibernate jar, and we want to do another dependency. And the <groupId> for this is org.hibernate, and the <artifactId> is going to be hibernate‑core. And then we just want to specify the version, and the version is 5.4.1.Final. I want to point out here that, in my opinion, line 29 violates a couple of industry‑common standards. They're not true industry standards, but the naming convention of the release is 5.4.1.Final. I personally think it should be a ‑Final, and I actually think that Final should be all uppercase. I actually prefer the word release, but this is what Hibernate chose. So, as I mentioned before, the version can be named specifically whatever that developer wants to call it. But if we go back to our Dependency Hierarchy tab, now look at all the dependencies it's pulled in. So we have antlr, byte‑buddy, classmate. We have all of these other dependencies down here that are resolve dependencies or transitive dependencies. You can see over in our Dependency Hierarchy which library pulled in which jar.



So hibernate‑commons‑annotations specifically pulled in jboss‑logging,



and the jaxb‑api pulled in the javax.activaton‑api. Jaxb‑runtime pulled in a half dozen different libraries that it needed. So you really start to see we just asked for hibernate‑core, and it's going to go pull the dependencies it needs and resolve any compilation errors or library dependencies that it needs to to pull in the correct set of libraries that your application needs. So it's a little bit of a complex example. Let's look at those three again. We had org.apache.commons that had no transitive dependencies. It just pulled its own library in. Then we threw the junit‑jupiter‑engine POM in here, and it pulled down two other POMs that had those libraries it needed. Then we pulled in the hibernate‑core that had its own unique naming convention, but look at all the transitive dependencies it pulled in. That's one reason I really like Spring STS or Eclipse for this, is this Dependency Hierarchy viewer really is pretty handy for you trying to troubleshoot which jar is pulling which library in.

# Summary

=>slides: Pg. 9

In this module we went through a bunch of the nuances with dependencies and we started looking at particular things like versions and that version numbers are really up to your corporate strategy. If you have a 1.0 release or a 1.0.0 you can name it final or release, it really doesn't matter. We had that hibernate example that was using final in their own unique way. The only one that has a unique meaning on the naming convention really, truthfully, is SNAPSHOT. SNAPSHOTS are always good to force you to download the latest version of the code, which is great if you're doing development and things are very volatile and you want to always be checking for the latest source code. You never want to go to production with any SNAPSHOTS in your application. We also looked at supported types and what they mean to your application. And 99% of the time you're probably just going to be using a jar dependency, but other than that what's very unique and kind of interesting is the POM dependency to where you can create a POM file that will be pulled in with all of the dependencies listed in it and use those inside of your application. We saw that POM dependency when we pulled in JUnit, and it pulled in two other POMs that had all the dependencies listed there. We also looked at transitive dependencies and how they are downloading. And the demo we saw that if we pulled down hibernate it would pull all of the transitive dependencies of hibernate down for us. In fact, there was even a conflict in there, and you can see that it resolved and looked for the newest version, which is generally what we want to do inside of our application. We also looked at scopes and how they allow us to compile our test code, but not include artifacts that shouldn't be packaged in our final resource. So when we include JUnit inside of our application, we can mark it as test scope, which is one of the most common ones that we typically scope, and it wouldn't be packaged into our final resource.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*