Contents

[Lesson 1. Intro 2](#_Toc72764311)

[Lesson 2. Installing 2](#_Toc72764312)

[Lesson 3. Plugins and MOJO 2](#_Toc72764313)

[Lesson 4. JVM arguments, Program arguments, Environment variables 2](#_Toc72764314)

[Lesson 5. Archetype plugin 3](#_Toc72764315)

[Lesson 6. POM Project Object Model 3](#_Toc72764316)

[Lesson 7. Effective POM 3](#_Toc72764317)

[Lesson 8. Dependency management 3](#_Toc72764318)

[Lesson 9. Dependency scope, Dependency plugin 4](#_Toc72764319)

[Lesson 10. Transitive dependencies, exclusions, optional dependencies 4](#_Toc72764320)

[Lesson 11. Project lifecycles. Clean lifecycle 4](#_Toc72764321)

[Lesson 12. Validate and compile phases 5](#_Toc72764322)

[Lesson 13. Test phase. Surefire plugin 6](#_Toc72764323)

[Lesson 14. Package phase 6](#_Toc72764324)

[Lesson 15. Packaging type war 7](#_Toc72764325)

[Lesson 16. Connecting of a goal to a phase 8](#_Toc72764326)

[Lesson 17. Verify phase. Failsafe plugin 8](#_Toc72764327)

[Lesson 18. Install phase 9](#_Toc72764328)

[Lesson 19. Deploy phase. Nexus 9](#_Toc72764329)

[Lesson 20. Site phase. Jacoco plugin 10](#_Toc72764330)

[Lesson 21. Multimodule project 11](#_Toc72764331)

[Lesson 22. PluginManagement, DependencyManagement, Best practices 12](#_Toc72764332)

[Lesson 23. Properties. Resource filtering 12](#_Toc72764333)

[Lesson 24. Profile 13](#_Toc72764334)

[Lesson 25. Executable jar 15](#_Toc72764335)

## Lesson 1. Intro

Before build tools projects were built by batch/shell scripts. It had drawbacks:

* Platform dependence (unix/windows required different scripts)
* Scripts from on project didn’t fit to another
* No uniform project structure (more onboarding time when you come on new project)

2000 – released Apache Ant which were more convenient than scripts.

2002 – released Apache Maven which was better because supplied uniformed project structure.

2007 – released Gradle which can use Groovy/Kotlin instead of xml.

-----------------------------------------------------------------------------------------------------------------------------------------------------------

## Lesson 2. Installing

About installing of environment variables (JAVA\_HOME, JDK) for unix/windows and about apache maven in the video. About installing of Maven Wrapper in the last video of the course.

-----------------------------------------------------------------------------------------------------------------------------------------------------------

## Lesson 3. Plugins and MOJO

Maven is just a set of plugins. Each plugin is a separate java project. Plugins contain commands (goals).

Goals are MOJO (Maven plain Old Java Object) – ordinary java classes with method execute(). So we can create own goals and plugins.

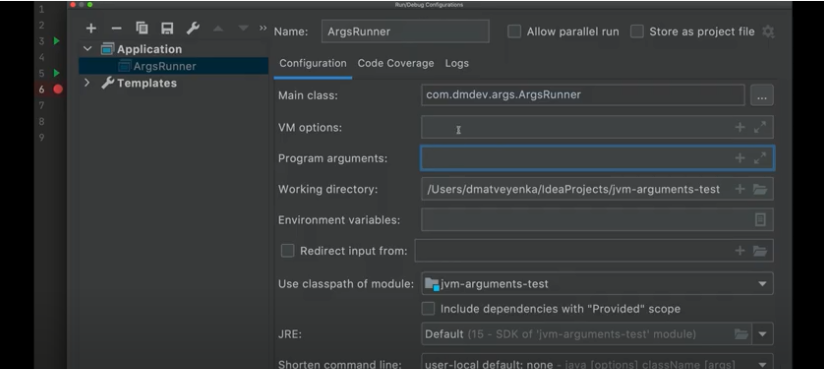
mvn plugin:goal (e.g. mvn compiler:compile)

Each plugin has goal “help”.

-----------------------------------------------------------------------------------------------------------------------------------------------------------

## Lesson 4. JVM arguments, Program arguments, Environment variables

IDEA->Run->Edit Configuration



Program arguments = arguments of main()

VM options = JVM arguments:

* User args Dkey=value
* Reserved args XkeyValue Xms512m – heap size during program start

XX… for tuning JIT compiler or garbage collector

To look at arguments passed to our program:

IDEA->DEBUG->evaluate expression->System.getProperties();

To look at environment variables:

IDEA->DEBUG->evaluate expression->System.getEnv(name:);

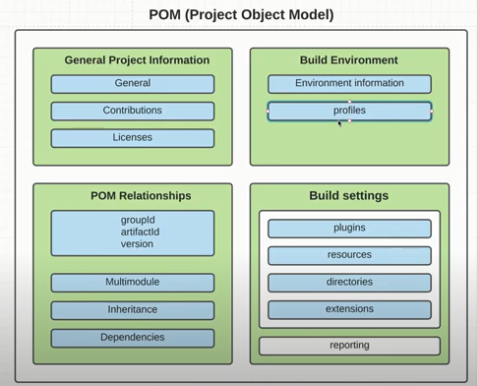
-----------------------------------------------------------------------------------------------------------------------------------------------------------

## Lesson 5. Archetype plugin

Generates project of required structure

-----------------------------------------------------------------------------------------------------------------------------------------------------------

## Lesson 6. POM Project Object Model



-----------------------------------------------------------------------------------------------------------------------------------------------------------

## Lesson 7. Effective POM

All POM files inherit super POM (like object in java) and parent POM files. To see result POM (effective) there is a goal:

mvn help:effective-pom

-----------------------------------------------------------------------------------------------------------------------------------------------------------

## Lesson 8. Dependency management

Classpath – directory where all our dependencies (.jar) are stored. They are not in our project but in maven local repository in our PC (~/.m2/repository). If we add a new dependency in our project maven first look for it in local repo and then in remote repo.

${maven}/conf/settings.xml – here we can change path to local repo (better not to do it)

-----------------------------------------------------------------------------------------------------------------------------------------------------------

## Lesson 9. Dependency scope, Dependency plugin

|  |  |  |
| --- | --- | --- |
| TYPE | DESCRIPTION | EXAMPLE |
| Compile (Default) | Dependency is needed to compile our project |  |
| Provided | By another dependency. | Jackarta.servlet-api is provided by Tomcat |
| Runtime | Dependency is needed in runtime. | Database driver |
| System | Jar is on our pc. | DO NOT USE AT ALL! |
| Test | Dependency is needed during test phase | Junit |

mvn dependency:analyze - shows unused or required dependencies

mvn dependency:tree -Dverbose - tree of dependencies

-----------------------------------------------------------------------------------------------------------------------------------------------------------

## Lesson 10. Transitive dependencies, exclusions, optional dependencies

Each dependency in our local maven repo has its own POM file. From it transitive dependencies are pulled by chain.

For instance spring dependencies pull a lot of transitive dependencies. Problems occur when those dependencies don’t fit by versions.

mvn dependency:tree –Dverbose

Sometimes in output of the goal above we can see that some dependencies are marked as “omitted for conflict with X.X.X RELEASE”.

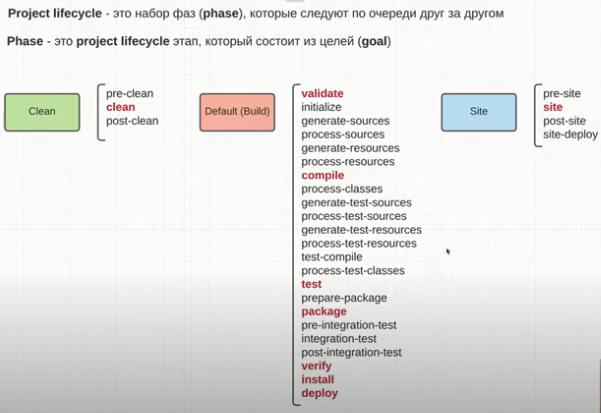
In such situation those dependencies win which are located higher in POM file. So effective POM depends on order of dependencies. In opposite to Maven in Gradle dependencies with higher version win.

Another way to solve dependency conflict is tag <exclusions></exclusions>.

Also we can make dependencies optional and anybody who connect our project as a dependency has to choose one of them and add to his POM. But better not to do it.

-----------------------------------------------------------------------------------------------------------------------------------------------------------

## Lesson 11. Project lifecycles. Clean lifecycle



Maven has 3 lifecycles. Each lifecycle consists of phases. Phase consists of goals.

Short description of phases:

* Clean – deleting folder target
* Validate – validate POMs
* Compile – generation .class files
* Test – unit testing
* Package – creating artefact (JAR/WAR)
* Verify – integration testing
* Install – copying of our artefact to local maven repo
* Deploy – pushing of our artefact to remote maven repo
* Site – generation of reports (documentation, test reports and so on)

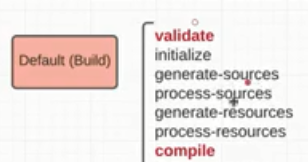
Set of goals on each phase depends on artefact type we choose, plugins and dependencies and so on.

When we work in IDEA we use not the Maven which is in our PATH variable, but maven chosen in:

Shift-shift/Maven home directory

-----------------------------------------------------------------------------------------------------------------------------------------------------------

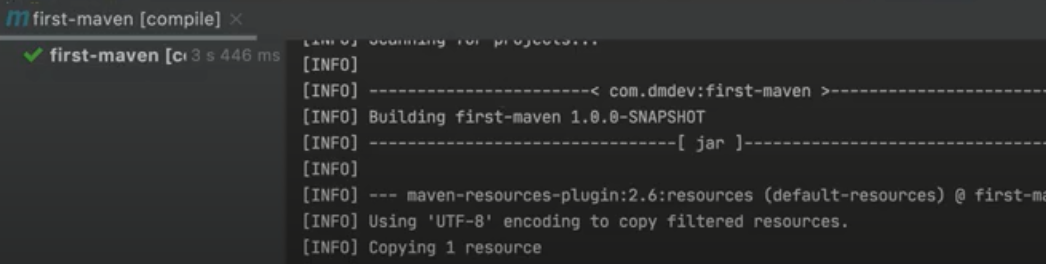
## Lesson 12. Validate and compile phases



All phases go consequently:

* Validate – validation of POMs
* Initialize – by default does nothing (we can override by connecting a plugin)
* Generate-sources – generator of .java (like QueryDSL) work on the phase
* Process sources – we can set vals of vars
* Generate-resources –
* Process resources – put all recourses from main to target folder
* Compile – generation .class files from .java

Which phases were passes we can see in output of terminal:

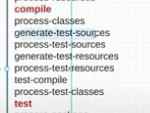


In section build we can override version and behavior of our plugins:

|  |  |
| --- | --- |
|  | <plugins> |
|  | <plugin> |
|  | <groupId>org.apache.maven.plugins</groupId> |
|  | <artifactId>maven-compiler-plugin</artifactId> |
|  | <version>3.8.1</version> |
|  | <configuration> |
|  | <source>15</source> |
|  | <target>15</target> |
|  | </configuration> |
|  | </plugin> |
|  | </plugins> |
|  | </build> |

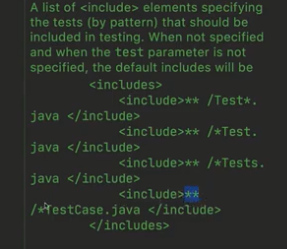
-----------------------------------------------------------------------------------------------------------------------------------------------------------

## Lesson 13. Test phase. Surefire plugin



* Process-classes – to change something in bytecode
* Others – clear from names

Surefire plugin runs our UNIT test and generates statistics. By default it runs tests with following names only:

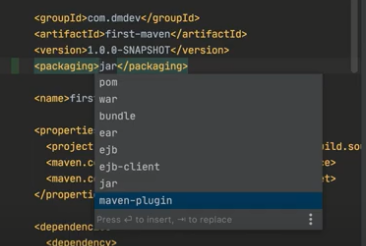


But this behavior can be overridden.

-----------------------------------------------------------------------------------------------------------------------------------------------------------

## Lesson 14. Package phase

By default projects are packaged to JAR, but there are other types (not only WAR):



We probably face only JAR, WAR, POM, Maven-plugin. Others are very rare.

NOTE: by default JAR is not executable, because it doesn’t contain required dependencies. It contains only our code. So it needs lib folder where required dependencies are stored. It can be added to WAR in IDEA

-----------------------------------------------------------------------------------------------------------------------------------------------------------

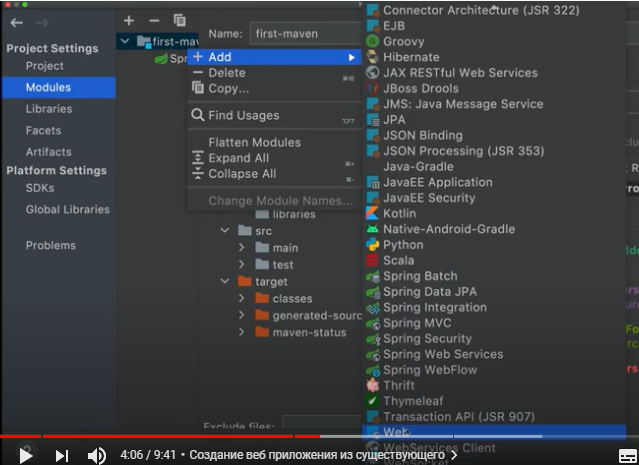
## Lesson 15. Packaging type war

1st way of creation:

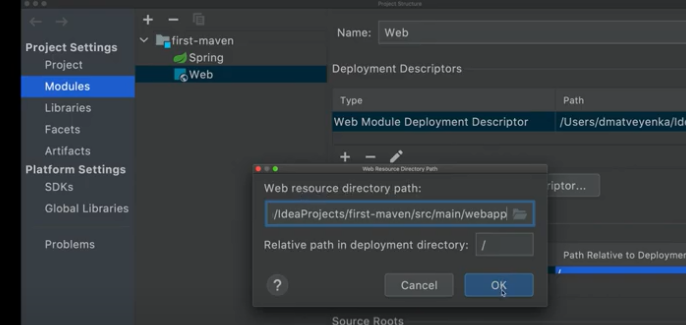
Maven-archetypre-webapp – better not to use (it generates old versions of web.xml and POM).

2nd way of creation:

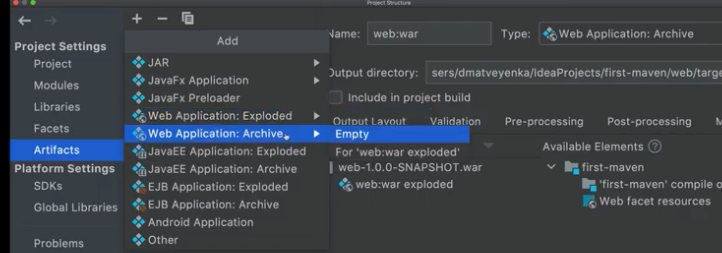
1. We create ordinary Maven project (JAR)
2. Then we add web module:



1. We move directory web to directory main and rename to webapp.
2. Go in POM and change packaging to war
3. We change web resource directory in settings (my IDEA did this automatically):



1. We add artefacts in settings (both Archive and Exploded). My Idea also did this automatically



That’s all. We can call package phase.

-----------------------------------------------------------------------------------------------------------------------------------------------------------

## Lesson 16. Connecting of a goal to a phase

We override plugin (which goal we want to use) in build section of POM.

Example of adding docker-compose.yml to target in phase validate is included.

-----------------------------------------------------------------------------------------------------------------------------------------------------------

## Lesson 17. Verify phase. Failsafe plugin

Failsafe plugin runs our INTEGRATION tests and generates statistics. By default it runs tests with prefixes and postfixes “IT”.

By default to verify phase no goal is connected. We override it in build section like in previous lesson.

Example as well is in attached project.

All reports (surefire and failsafe) are in target folder.

-----------------------------------------------------------------------------------------------------------------------------------------------------------

## Lesson 18. Install phase

On the phase our artifact is saved in our local Maven repo.

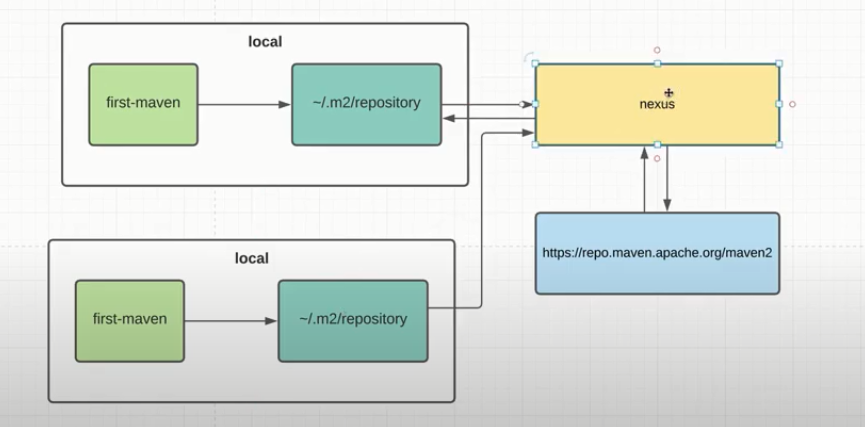
It’s a good practice to override all Maven plugins, because by default they have old versions. Overridden in project.

-----------------------------------------------------------------------------------------------------------------------------------------------------------

## Lesson 19. Deploy phase. Nexus

We don’t have rights to deploy our artifacts to remote Maven repository.

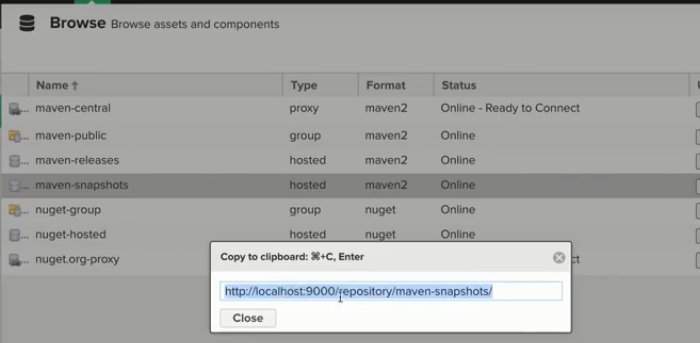
The most widespread available repository for Maven is Nexus. Nexus is also used like cache for dependencies.



Programmers pull dependencies from Nexus. If dependency is absent in Nexus, Nexus pulls it from remote Maven repository.

The simplest way to deploy Nexus is to run it in Docker container.

Nexus has separate URLs for snapshots, releases and others. We use those URLs in our POM file.



Our POM changes:

|  |  |
| --- | --- |
|  | <!-- ~/.m2/settings.xml--> |
|  | <distributionManagement> |
|  | <snapshotRepository> |
|  | <id>nexus</id> |
|  | <url>http://localhost:9000/repository/maven-snapshots/</url> |
|  | </snapshotRepository> |
|  | <repository> |
|  | <id>nexus</id> |
|  | <url>http://localhost:9000/repository/maven-releases/</url> |
|  | </repository> |
|  | </distributionManagement> |

Also we have to set id in settings.xml. Settings.xml is a file where common Maven settings are configured (logins, passwords and so on).

-----------------------------------------------------------------------------------------------------------------------------------------------------------

## Lesson 20. Site phase. Jacoco plugin

It’s phase for generation of documentation (for instance about our classes) and reports (for instance about tests).

Generation of documentation works out of box. We just have to override needed site plugins in build section:

|  |
| --- |
|  |
|  | <plugin> |
|  | <groupId>org.apache.maven.plugins</groupId> |
|  | <artifactId>maven-site-plugin</artifactId> |
|  | <version>3.9.1</version> |
|  | </plugin> |
|  | <plugin> |
|  | <groupId>org.apache.maven.plugins</groupId> |
|  | <artifactId>maven-project-info-reports-plugin</artifactId> |
|  | <version>3.1.1</version> |
|  | </plugin> |

And then just call site phase. Reports will be stored in target/site directory. Example of report:



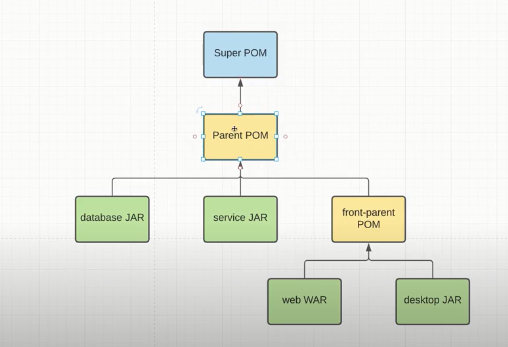
Generation of test also is not difficult. We add Jacoco plugin to section Build of POM. Example is below:

|  |
| --- |
|  |
|  | <plugin> |
|  | <groupId>org.jacoco</groupId> |
|  | <artifactId>jacoco-maven-plugin</artifactId> |
|  | <version>0.8.6</version> |
|  | <executions> |
|  | <execution> |
|  | <id>prepare-agent</id> |
|  | <goals> |
|  | <goal>prepare-agent</goal> |
|  | </goals> |
|  | </execution> |
|  | <execution> |
|  | <id>generate-jacoco-report</id> |
|  | <goals> |
|  | <goal>report</goal> |
|  | </goals> |
|  | <phase>prepare-package</phase> |
|  | </execution> |
|  | </executions> |
|  | </plugin> |

Jacoco shows test coverage of our classes.

-----------------------------------------------------------------------------------------------------------------------------------------------------------

## Lesson 21. Multimodule project

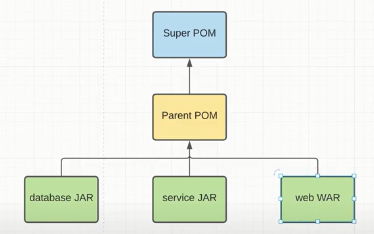


Real applications are much bigger then what we do on courses. Multimodule structure is convenient for them.

<packaging>pom</packaging>

This packaging type is for inheritance.

In the course we make such a structure:



We just add new modules and delete src dir from first-maven-project, because it automatically becomes “pom” packaging type.

-----------------------------------------------------------------------------------------------------------------------------------------------------------

## Lesson 22. PluginManagement, DependencyManagement, Best practices

Best practices:

* Put all dependencies of parent project in dependencyManagement tag (they won’t be inherited anymore)
* Override dependencies in child project without scope and version (they should be defined in parent project)
* Put a blank line between dependencies with different scope
* Put all plugins of parent project in pluginManagement tag
* Put version of dependency group into property and use through expression language

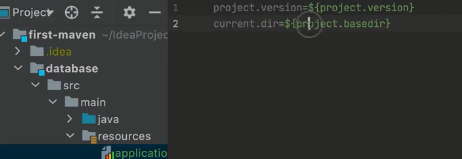
-----------------------------------------------------------------------------------------------------------------------------------------------------------

## Lesson 23. Properties. Resource filtering

There are 5 groups of properties:

* project.\* - properties of our root tag project (like groupId and others)
* settings.\* - properties of setting.xml (used very seldom)
* ${\*} - properties that we define in tag properties
* env.\* - environment variables
* -D - system.getProperty

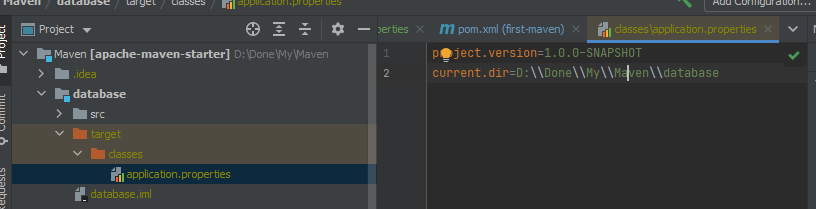
We can transfer properties from POM to project through expression language:



But by default Maven will not put properties there. We have to set filtering on for resource folder in POM:



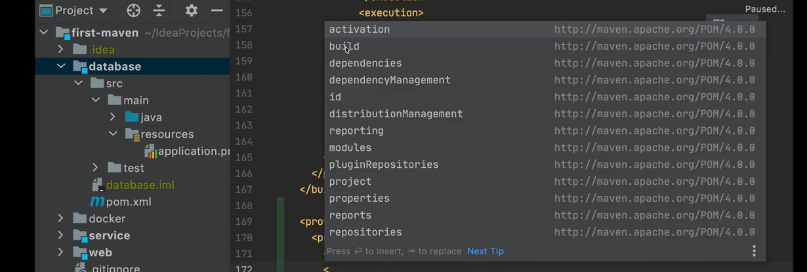
After calling resources:recources values will be putted in propertie file in target folder:



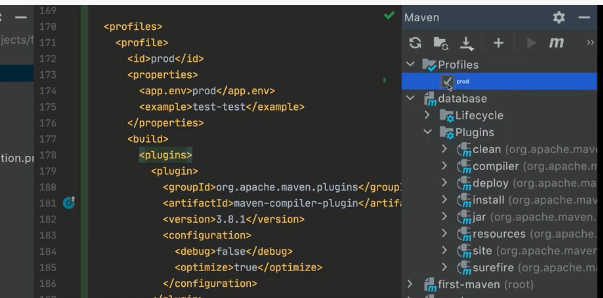
-----------------------------------------------------------------------------------------------------------------------------------------------------------

## Lesson 24. Profile

In profile we can override a lot of things:



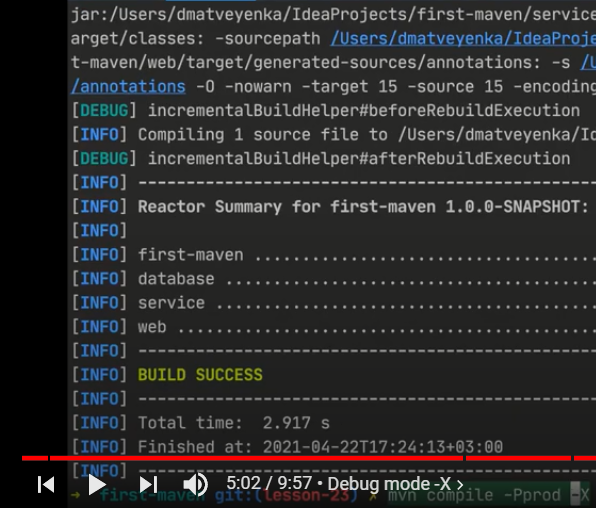
We can choose a profile in IDEA:



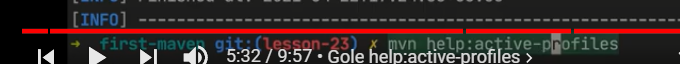
If we want to choose profiles through terminal we separate them by coma:



Flag –X starts Maven in debug mode. In debug mode we can see additional info (like settings of plugins):



Help can show us active profiles:



In profile section there is tag acticvation. Activation could be of following types:

* <by default>
* <os> very seldom used in practice
* <jdk>
* <file> (exist or missing)
* <property> -D
* More than one from previous its logical AND

-----------------------------------------------------------------------------------------------------------------------------------------------------------

## Lesson 25. Executable jar

Maven package phase creates packaging type artifacts. We can use the like dependencies in other modules or projects, or just run as ordinary java app. In the lesson we’ll find out how to make it correctly and why we need MANIFEST.MF file.

Let’s try to compile service module. We get error because it uses database module as dependency, so we have to put the artifact in local repo first (install phase for database module). Then compile phase for service module will be successful. And we can run compiled class “java target/classes/com/practice/AppRunner.class”.

But in practice nobody runs java apps like that. Applications are usually collected to jar files (package phase). So let’s call package file. If we try to run it “java –jar target/service-1.0.0-SNAPSHOT.jar” we’ll get error “no main manifest attribute”. That because we need to specify a path to method main in manifest file.

It’s done in POM file by overriding maven-jar-pluggin:

|  |
| --- |
| <build> |
|  | <plugins> |
|  | <plugin> |
|  | <groupId>org.apache.maven.plugins</groupId> |
|  | <artifactId>maven-jar-plugin</artifactId> |
|  | <version>3.2.0</version> |
|  | <configuration> |
|  | <archive> |
|  | <manifest> |
|  | <mainClass>com.dmdev.service.AppRunner</mainClass> |
|  | </manifest> |
|  | </archive> |
|  | </configuration> |
|  | </plugin> |
|  | </plugins> |
|  | </build> |

After that we can package again and run our jar file.