Build tools

Agenda

1. What are build tools for?
2. Maven
    - What is?
    - Setup/Local & remote repositories
    - Understanding the Project Object Model (POM)
    - Dependencies
    - Phases
    - Multi Module
3. Maven conventions

Build Tools

Software that automates the process of compiling, testing, packaging and deploying source code in the most efficient manner, making the process repeatable less error prone

Some worthy mentions (for a Java Dev)

- MAKE
- Apache Ant
- **Apache Maven** *** **(We'll be focusing here)**
- Apache Ivy
- Gradle

Other mentions (non-Java)

- NPM

Automates

Gives each developer the ability to build and run on their machine, making the process repeatable and portable

Dependency Management

Automates the process of downloading and managing dependencies. No need to keep JAR files in the lib folder. Ensures correct versions

Correct execution order

Helps large projects execute build commands in the correct order based on dependencies (See multi-module for example)

Saves time

Certain build tools can even run commands in parallel helping you save time

Precondition for Continuous Integration/Delivery

Continuous Integration (CI) is achieved by running automated builds several times a day, for each code check-in done to a shared repository. You can't even think on CI (or even Continuous Delivery) without an automated build

Maven

- Build automation tool used for Java projects (nowadays can be used for other languages as well)
- Addresses two aspects: How software is built and its dependencies
- Makes use of a local cache for artifact locating. If it can't find it, it downloads it and stores it
- Proposes an opinionated but simple approach to project folder structure

Not recommended (currently)

Having your own maven installation (you can install it via https://sdkman.io/) in case you need it.

Still, if someone is creating a project FROM SCRATCH, will probably need to have maven installed

- Required a JRE/JDK available
- Use the maven wrapper (mvnw for *nix based or Windows) whenever possible. This is a distributed binary that guarantees the maven version to be used
- Use the default settings and default local repo location whenever possible
- Minimum required unit is a POM file
- Optionally: you can generate from archetypes (out of scope)

Local repository

This is the place where all dependencies/artifacts are stored within your machine.
In simple words, is a folder that will hold each and every dependency your project depends on

- Located by default in ${user.home}/.m2/repository (*)
- Holds a folder with the name of the artifact (with package structure)
- Each artifact folder holds several subfolders with the different versions of a given artifact
- Each subfolder contains a set of files including the actual binary (typically a JAR)(**) and the POM descriptor

(*) You can change this but, trust us, it's not worth it.
(**) Other types of packaging are supported

Central repository

This is the place on the internet where all **dependencies/artifacts** with a proper licensing policy are stored (Open Source typically). Provided by the maven community.

You don't need to configure anything to have access to the Central repository. However, you may need to configure certain Mirrors in case you have limited access

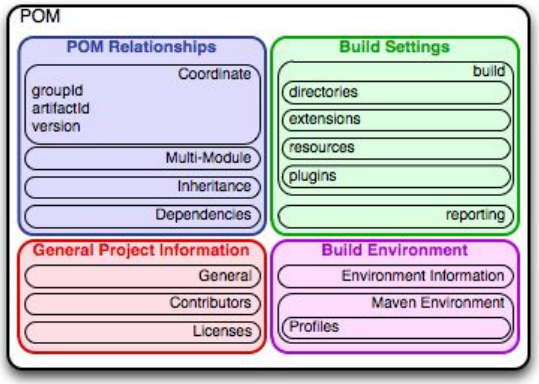
- Located at https://repo1.maven.org/maven2/
- Indexes all artifacts distributed by the community
- Other private repositories are indexed at https://mvnrepository.com
- For instance, you can see what artifacts are available for *org.springframework* at https://repo1.maven.org/maven2/org/springframework/
- Or better, you can search at https://search.maven.org/ or https://mvnrepository.com

Remote/Custom repository

This is the place on the internet or an intranet where all **dependencies/artifacts** relevant to an organization are stored.

You need to configure access to these repositories within your POM files or through a *settings.xml* file inside your local repository folder

- Organizations with artifact versioning will typically host their own repositories
- Some Cloud providers already provide custom repositories (Like Amazon or Azure)
- If there's any authentication policy, it will be handled at POM level (plugin configuration) or via the global *settings.xml* file



Contains four categories of description and configuration

- General project information
- Build settings
- Build environment
- POM relationships

All POMs inherit from the Super POM, which is part of the maven installation. You can say the POM is like, in Java, the *java.lang.Object* class

Simplest POM

```
<project>
    <modelVersion>4.0.0</modelVersion>

<groupId>org.sonatype.mavenbook.ch08</groupId>
    <artifactId>simplest-project</artifactId>
    <version>1</version>
</project>
```

Artifact: simplest-project-1.jar
Location:
${m2.home}/org/sonatype/mavenbook/ch08

The simplest POM requires the following information

- *groupId* (like a java package)
- *artifactId* (the name of your artifact)
- *version* (<major>.<minor>.<incremental>-<qualifier>)
    - Suggested but not enforced
- ** *modelVersion* ** (use always version 4.0.0)

Most common sections

```
<dependencies>
    …
</dependencies>

<dependencyManagement>
    …
</dependencyManagement>

<properties>
    …
</properties>

<parent>
    …
</parent>

<build>
    …
</build>
```

Among others

- *dependencies*: Place where dependencies are declared
- *dependencyManagement*: Place where dependencies are declared in a centralized manner
- *properties: Place where properties are declared to be used across this POM file or any child*
- Parent: Place to declare the module that acts as the parent of the current module. The module inherits from the parent
- Build: Place to declare specific plugins and customizations/configurations to plugins

Example

```
<project>
    ...
    <dependencies>
        <dependency>
            <groupId>org.codehaus.xfire</groupId>
            <artifactId>xfire-java5</artifactId>
            <version>1.2.5</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.4</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
    ...
</project>
```

- A dependency declares a relationship between this module and the artifact declared inside the *dependency* section. Requires *at least* groupId and artifactId. Version is required but can be taken from dependencyManagement section or parent
- Dependencies have a scope: compile, provided, runtime, test, system (avoid this last one)
- Dependencies can be optional
- Some dependencies can be transitive (*)

(*) Remember the Log4shell bug?

Three standard lifecycles

- **Clean**
- **Default**
- **Site**

- A lifecycle is an organized sequence of phases that exist to give order to a set of goals
- A lifecycle phase is bound to a particular goal (which is, the execution of a defined plugin)
- **Clean** is the simplest and consists of three lifecycle phases. As its name suggests, it cleans the build state, deleting any resources that were compiled or generated for building purposes
- **Site** is another one that will help generate project documentation
- The **default** lifecycle is the one most users will be familiar with

Three standard lifecycles

- **Clean**
- **Default**
- **Site**

- A lifecycle is an organized sequence of phases that exist to give order to a set of goals
- A lifecycle phase is bound to a particular goal (which is, the execution of a defined plugin)
- **Clean** is the simplest and consists of three lifecycle phases. As its name suggests, it cleans the build state, deleting any resources that were compiled or generated for building purposes
- **Site** is another one that will help generate project documentation
- The **default** lifecycle is the one most users will be familiar with, responsible for build and deployment

Some Maven phases

- **validate**
- **compile**
- **test-compile**
- **test**
- **package**
- **integration-test**
- **install**
- **deploy**

- Phases are executed in a specific order. This means that if we run a specific phase using the command `mvn <PHASE>`

  This won't only execute the specified phase but all the preceding phases as well

- Each phase is a sequence of goals, and each goal is responsible for a specific task

Examples

- **compile**
- **test-compile**
- **test**
- **package**
- **install**

So, take into consideration that each phase executes a set of goals. All of this is automated for you

Here are some of the phases and default goals bound to them:

- *compiler:compile* – the *compile* goal from the *compiler* plugin is bound to the *compile* phase
- *compiler:testCompile* is bound to the *test-compile* phase
- *surefire:test* is bound to *test* phase
- *install:install* is bound to *install* phase
- *jar:jar* and *war:war* is bound to *package* phase

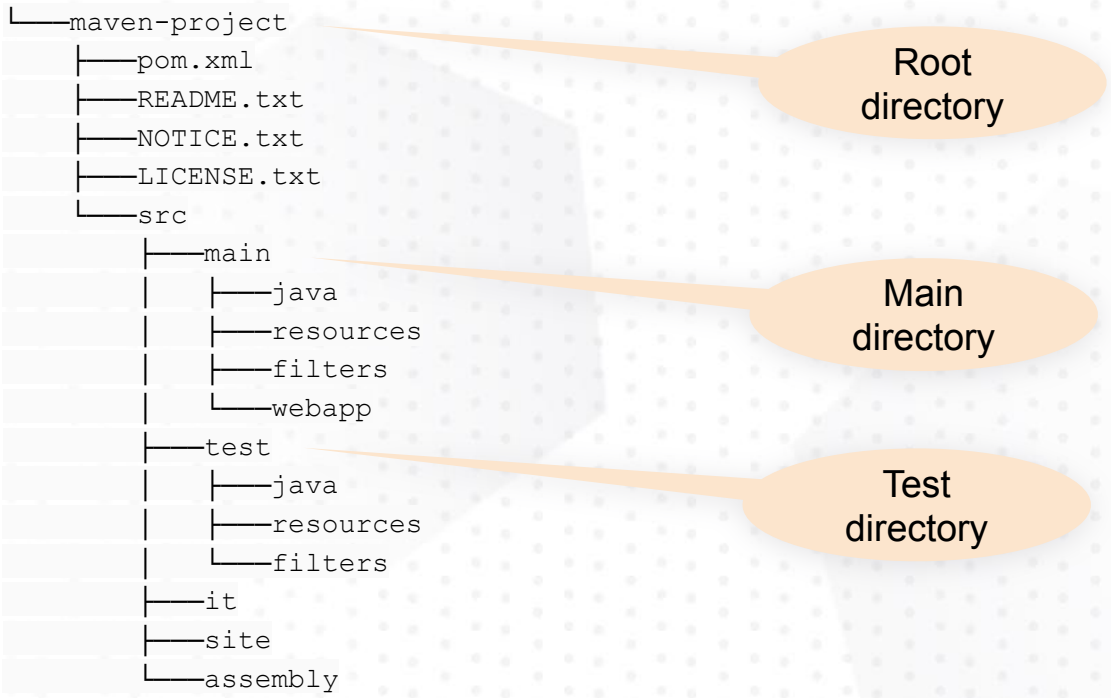
Back to the parent POM

```
Parent POM file
… //Module section
<modules>
    <module>core</module>
    <module>service</module>
    <module>webapp</module>
</modules>
 …
```

```
//Sub module
<parent>
  <artifactId>parent-project</artifactId>
  <groupId>com.codurance</groupId>
  <version>1.0-SNAPSHOT</version>
</parent>
```

- Parent POM is used as an aggregator that manages a group of submodules
- In *most cases* the aggregator is located in the project's root directory and MUST have packaging of type *pom*.
- By building the project through the aggregator POM, each project that has a packaging type different from *pom* will result in a built archive file
- Parent POM must declare a set of *submodules* if you want to build in one go
- Submodules must declare a parent POM from which they inherit

Maven conventions


maven-project
pom.xml
README.txt
NOTICE.txt
LICENSE.txt
src
main
java
resources
filters
webapp
test
java
resources
filters
it
site
assembly
Root directory
Main directory
Test directory

Convention over configuration

Proposes that: Systems, libraries and frameworks should assume reasonable defaults. This means, without requiring unnecessary configuration, systems should *just work*

- Incorporates sensible default behavior.
- Folder structure assumed by default, but highly customizable
- Default generated folders for compiled sources
- Default JAR packaging type
- In summary, “opinionated”

Exercises

Find the exercises for this module in the following link

Thank you

If you have any questions,
please get in touch.