# buildingtool

### Voorbeeld

### Te gebruiken bij:

-

### Eigenschappen:

-

### Info

#### Build tools en dependencies

Hoewel je tijdens deze leerlijn nog voornamelijk in je eentje aan projecten werkt, werk je in het bedrijfsleven eigenlijk zelden alleen. In de praktijk ontwikkel je software samen met meerdere collega-developers in een team - en vaak ook nog in samenwerking met andere development-teams! Om die onderlinge samenwerking mogelijk te maken, werken we met een breed scala aan tools:

1. **Communicatie:** Slack, Teams, Meet, Zoom, etc.
2. **Versiebeheer**: Github, Gitlab, Bitbucket, etc.
3. **Operatie**: Amazon Web Services, Google Cloud, Microsoft Azure, Oracle Cloud, IBM Cloud, etc.
4. **Integratie**: Jenkins, Tekton, CircleCI, Github Actions, Gitlab Runners, etc.
5. **Deployment**: ArgoCD, Ansible, Terraform, etc.

Om al deze processen te integreren gebruiken we **Build Tooling**.

Build tooling is een set aan tools die alle bovengenoemde randprocessen voor ons voor ons automatiseert. Dit zorgt er niet alleen voor dat jij je als developer kunt focussen op het belangrijkste - het bouwen van de software! - maar ook dat dit bij iedere developer op dezelfde manier gebeurt.

Dankzij build tooling kunnen we onze code namelijk met simpele commando's

1. compileren (draaien),
2. verpakken (uitvoerbaar maken voor andere systemen) en
3. implementeren (in gebruik nemen) in verschillende omgevingen.

Afbeelding met tekst, schermopname, Lettertype, diagram

Automatisch gegenereerde beschrijvingBuild tooling speelt dus een hele grote rol: kijk maar naar onze Application Lifecyle:

In drie van de vijf fases worden we door build tooling ondersteunt.

We beheren er de **dependencies** in ons project mee.

een library.

Dat klinkt heel fancy, maar een library is niets meer dan een geïsoleerd bundeltje Java code. Dit bundeltje kan één regel code tot tientallen classes bevatten. Die classes zijn gecomprimeerd, ofwel ingepakt, tot een JAR: een **J**ava **AR**chive.

#### wat zijn dependencies eigenlijk?

#### Maven in gebruik

Wanneer we gebruik maken van Maven, doen we dat doormiddel van commando's die we in de terminal uitvoeren. Hierbij is het belangrijk om onderscheid te maken tussen een project met een lokale versie van Maven, of een project met een meegeleverde Maven Wrapper. Wanneer je gebruik maakt van jouw eigen versie van Maven, begint ieder commando met:

mvn

Dit commando doet zelf nog niets, maar wordt aangevuld met een actie. Dit zul je later in deze paragraaf tegenkomen.

Indien je gebruik maakt van een Maven Wrapper is het belangrijk het commando aan te passen op basis van het besturingssysteem van jouw computer. Dan begint ieder commando met:

// MacOS en Linux:

./mvnw

​

// Windows:

mvnw.cmd

Het basiscommando kunnen we vervolgens uitbreiden om verschillende soorten acties uit te voeren met Maven. Voor we verder in deze commando's duiken, is het belangrijk om de verschillen tussen onderstaande begrippen goed te begrijpen.

##### compileren

Wanneer we het hebben over het **compileren** van de code, betekent dit enkel het omzetten van de door ons geschreven code naar code die uitvoerbaar is voor de JVM (Java Virtual Machine).   
Hierbij worden verschillende target-bestanden gecreëerd.

##### runnen

Dit is niet hetzelfde als het **runnen** van de code.   
Wanneer we de code runnen, betekent dit dat de omgezette code uit de target-map wordt uitgevoerd door de JVM en het project 'draait'.

##### builden

Ten slotte spreken we vaak over het **builden** van een project: hiermee wordt het gehele proces van compileren, testen, verpakken en deployen bedoelt.

Laten we eens kijken naar de individuele commando's:

validate wordt gebruikt om de pom.xml op structuurfouten te controleren. Hierover later meer;

compile wordt gebruikt om alle Java classes in het project om te zetten naar JVM bytecode;

test wordt gebruikt om alle unittests in het project uit te voeren. Indien compileren nodig is, wordt die taak eerst uitgevoerd;

package wordt gebruikt om het product in een JAR of een ander formaat (zoals in de pom.xml aangegeven met <packaging>...</packaging>, hierover later meer) te verpakken;

verify zorgt voor validatie van de Maven configuratie, het compileren van de code én het uitvoeren van alle unittests in het project;

deploy om het geteste, gecompileerde en verpakte product (de build) naar een (GitHub) repository te sturen.   
In de meeste gevallen zorgt dit er dan ook voor dat het project online komt te staan voor de eindgebruiker, vanwege externe koppelingen met GitHub.

clean wordt gebruikt om de target-map, waar onder andere alle omgezette JVM bytecode wordt geplaatst, helemaal leeg te maken zodat je met een schone lei kunt beginnen. Dit wordt over het algemeen alleen gebruikt wanneer je vindt dat de resultaten van het verify commando niet naar verwachting zijn;

Maven geeft ons dus een hoop mogelijkheden die we samen met het basiscommando kunnen gebruiken in de terminal. Om een voorbeeld te geven, het compileren van de code doen we met een lokale versie van Maven als volgt:

mvn package

**Let op:** wanneer je Maven als onderdeel van IntelliJ gebruikt (je hebt het dus niet handmatig op je OS geïnstalleerd) druk je niet op Enter, maar op CTRL + Enter (Windows) of CMD + Enter (Mac). Als je alleen op Enter drukt, zal dit commando niet herkend worden.

Naast de basisprocessen is het ook mogelijk om onze commando's te voorzien van extra opties of configuraties. Dit zijn commando's die we aan onze standaard commando's vast kunnen plakken:

Afbeelding met tekst, schermopname, Lettertype, nummer

Automatisch gegenereerde beschrijving

Op deze manier kunnen we tijdens het compilen bijvoorbeeld aangeven dat we geen kleuren, maar wel de Maven-, JDK- en Operating System-versies willen zien in de terminal:

mvn compile --batch-mode --show-version

De volgorde waarop je de opties achter het commando plakt, is niet van belang. Indien het voor jouw project belangrijk is om sommige opties altijd mee te geven, is het beter om deze in een configuratiebestand te plaatsen zodat ze automatisch worden meegenomen. Dit doe je normaliter in een maven.config in de .mvn map.

##### Configuratie

Je hebt inmiddels gelezen dat Maven een ontzettend krachtige tool is om onze projecten mee te builden en te beheren. Echter, de manier waarop Maven een project moet compileren, testen en deployen komt niet zomaar uit de lucht vallen. Hiervoor gebruikt Maven altijd een **POM-bestand**. POM staat voor Project Object Model en is een soort configuratie-bestand waarin alle instellingen en informatie over het project staan op geslagen. Zonder POM in jouw project zou Maven geen idee hebben wat er zou moeten gebeuren wanneer je vraagt om het project te compileren. Het POM-bestand moet daarom altijd op root-niveau in jouw project aanwezig zijn:

├── src

│ └── main

│   ├── java

│   └── resources

│ └── test

│   ├── java

│  └── resources

└── pom.xml

Indien er een Maven Wrapper aanwezig is, voegt deze wrapper een tweetal uitvoerbare bestanden én een mvn-map toe aan het project. Deze zul je in de huiswerkopdrachten al vaker zijn tegengekomen:

├── mvn

│ └── wrapper

│   ├── maven-wrapper.jar

│   └── maven-wrapper.properties

├── src

│ └── main

│   ├── java

│   └── resources

│ └── test

│   ├── java

│  └── resources

├── mvnw

├── mvnw.cmd

└── pom.xml

Een POM heeft de bestandsextensie .xml. **XML** is een mark-up taal die wordt gebruikt om data op te slaan en te delen. Het staat voor E**x**tensible **M**arkup **L**anguage en is ontworpen om gegevens te verzenden en te ontvangen tussen verschillende systemen. Het maakt gebruik van tags die verschillende objecten en data identificeren en lijkt een beetje om het welbekende **HTML**: Hypertext Markup Language.

De meest minimale pom.xml bevat alleen wat algemene informatie over het project:

<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">

 <modelVersion>4.0.0</modelVersion>

 <groupId>group-id</groupId>

 <artifactId>project-id</artifactId>

 <version>1.0</version>

</project>

Zoals je kunt zien is deze informatie ingesloten in tags. In de <version>-tag staat bijvoorbeeld wat de versie van ons project is, en in de <modelVersion>-tag staat welke versie van Maven gebruikt wordt in dit project. De onderdelen <groupId>, <artifactId> en <version> zijn verplichte onderdelen in een POM. Dit noem je daarom ook wel de **coördinaten** van een Maven project.

Wanneer je jouw project beschikbaar zou willen maken als open source library, zul je het project moeten deployen naar de officiële Maven repository. De coördinaten in jouw pom.xml zorgen ervoor dat het **artifact** - het opgeleverde project - vindbaar is tussen alle andere open source projecten die Maven rijk is.\*

Een normaal POM-bestand is echter altijd een stuk uitgebreider. Al deze onderdelen zullen we in de volgende paragrafen stapsgewijs toevoegen.

##### POM: projectinformatie

In het eerste gedeelte van de POM staat normaliter alle informatie over het project. We noemen onze projecten ook wel producten, gezien de bedoeling van deployment is dat eindgebruikers gebruik kunnen maken van onze applicatie.

<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">

​

 <modelVersion>4.0.0</modelVersion>

​

  <groupId>nl.novi</groupId>

  <artifactId>eindopdracht</artifactId>

  <version>1.2.4</version>

  <packaging>jar</packaging>

  <name>Mijn eindopdracht product</name>

  <description>Een applicatie die aan alle verwachtingen van de Integrale eindtoets Backend voldoet!</description>

  <url>https://www.eindopdracht-novi.nl</url>

  <inceptionYear>2023</inceptionYear>

</project>

In de <packaging>-tag geef je aan welke vorm van packaging gebruikt wordt:

* jar (is de standaard )
* war
* ear
* pom
* rar
* maven-plugin
* oejb

##### **Gedeelte informatie**

Hoewel je hier tijdens de leerlijn nog weinig mee te maken krijgt, komt het in de praktijk vaak voor dat een bedrijf meerdere kleine producten maakt die onderdeel zijn van één overkoepelend product(Netflix**)**. Dit noem je ook wel **microservices.**

De Netflix applicatie zoals jij 'm kent *gebruik* van een ander product dat verantwoordelijk is voor de opslag van alle films en series op het platform.  
Deze microservice beheert de database en geeft de applicatie toegang tot deze resources.

Een andere microservice is een product dat enkel en alleen berekent welke andere films je leuk zou kunnen vinden

###### Vanuit microservice

veel van de POM-instellingen voor die microservices hetzelfde zullen zijn. Daarom is het mogelijk om een **parent** op te nemen in de POM die allerlei configuraties importeert

<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">

​

 <modelVersion>4.0.0</modelVersion>

 <parent>

  <groupId>com.netflix</groupId>

  <artifactId>shared</artifactId>

  <version>1.2.5</version>

 </parent>

​

 <artifactId>recommendation-engine</artifactId>

 <version>1.2.4</version>

</project>

Omdat we nu alle instellingen overnemen uit de POM van onze denkbeeldige Netflix parent, hoeven we zelf bijna niks meer toe te voegen. Wel is het mogelijk om alle configuraties die we overnemen uit deze parent te *overschrijven*.

Dit hebben we dan ook gedaan voor de <artifactId>- en <version>-tag,

gezien deze natuurlijk wel uniek zijn voor deze specifieke microservice.

###### Vanuit de parent

Andersom zou je ook via de parent kunnen aangeven dat je deze POM-configuratie wil delen met verschillende sub-projecten of microservices. Dit doen we middels de <modules>-tag:

<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">

 <modelVersion>4.0.0</modelVersion>

 <groupId>com.netflix</groupId>

 <artifactId>parent</artifactId>

 <version>2.0</version>

 <packaging>pom</packaging>

 <modules>

  <module>recommendation-engine</module>

  <module>data-storage</module>

  <module>content-encoder</module>

 </modules>

</project>

##### POM: dependencies

Maven beheert alle libraries op één centrale verzamelplek: **Maven Central**. Dit is een repository waarin de open-source Java-projecten worden bewaard.

Het wordt door veel ontwikkelaars gebruikt om hun projecten en libraries te publiceren en te delen met anderen. Ook jij kunt een library bouwen en deze toevoegen aan Maven Central!

Wanneer we een library aan ons project willen toevoegen als dependency, zorgt Maven ervoor dat die library uit de Maven Central repository gedownload wordt wanneer je deze toevoegt aan jouw pom.xml.

Daarom zijn die coördinaten zo belangrijk:

Wanneer je hebt besloten een dependency te willen gebruiken, zet je deze tussen de <dependencies>-tag in de POM. Ieder <dependency>-element moet minimaal de volgende gegevens bevatten:

* *groupId* (het bedrijf of de persoon die de library gemaakt heeft)
* *artifactId* (de naam van de library)
* *versienummer* (wil je de meest recente versie of eentje van twee jaar terug?)

<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">

​

<modelVersion>4.0.0</modelVersion>

​

<!-- De projectinformatie staat hier.. -->

<!-- De dependencies staan hier.. -->

<dependencies>

<dependency>

<groupId>nl.novi</groupId>

<artifactId>bibliotheek</artifactId>

<version>1.3.5</version>

<scope>test</scope>

<type>jar</type>

<optional>false</optional>

</dependency>

</dependencies>

</project>

###### **Scope**

De <scope>-tag geeft aan binnen welke context we deze dependency willen gebruiken. Dit heeft allemaal te maken met de lifecylefases van Maven. De verschillende soorten scopes zijn:

* Compile: dependencies met deze scope zijn in alle scopes beschikbaar;
* Test: deze dependency is alleen beschikbaar tijdens het uitvoeren van unittests. Een goed voorbeeld daarvan is de JUnit library, die wij ook zullen gaan gebruiken.
* Runtime: deze dependency is alleen beschikbaar tijdens het uitvoeren van de applicatie. En dan niet als we ons project zelf op onze laptop draaien, maar wanneer het product gedeployt is en op een server draait. "In productie", noemen we dit ook wel. Een voorbeeld hiervan is een library die alle uitgaande en inkomende API calls logt, zodat wij als developers kunnen welke endpoints van onze API bibliotheek het populairst zijn.
* Provided: deze dependency wordt via een andere weg beschikbaar gemaakt, bijvoorbeeld via het project dat deze provided dependency als parent heeft;
* System: vrijwel identiek aan provided, maar nu wordt er vanuit gegaan dat de JAR altijd beschikbaar is, zonder download vanaf een repository. Wordt deze scope gebruikt, dan moet ook een systemPath opgegeven worden!

###### **Type**

Net als dat wij de mogelijkheid hebben ons project in verschillende formaten te verpakken met de <packaging>-tag, kunnen we met de <type>-tag aangeven in welk bestandsformaat we de library willen gebruiken. Wanneer we geen type opgeven, wordt er aangenomen dat je een JAR (.jar) wil hebben. Dit is dusdanig gebruikelijk dat we er rustig vanuit kunnen gaan dat je dit veld bijna nooit hoeft te gebruiken.

###### **Optional**

De <optional>-tag verwacht altijd een true of false waarde: een boolean. Dit noemen we daarom ookwel een **flag**: het is een instelling die we aan of uit kunnen zetten. Soms is het zo dat een library alleen nuttig is tijdens het compileren, maar niet tijdens runtime. In dat geval kunnen we deze dependency flaggen, waarmee we aangeven dat 'ie niet vereist is om het project te kunnen uitvoeren.