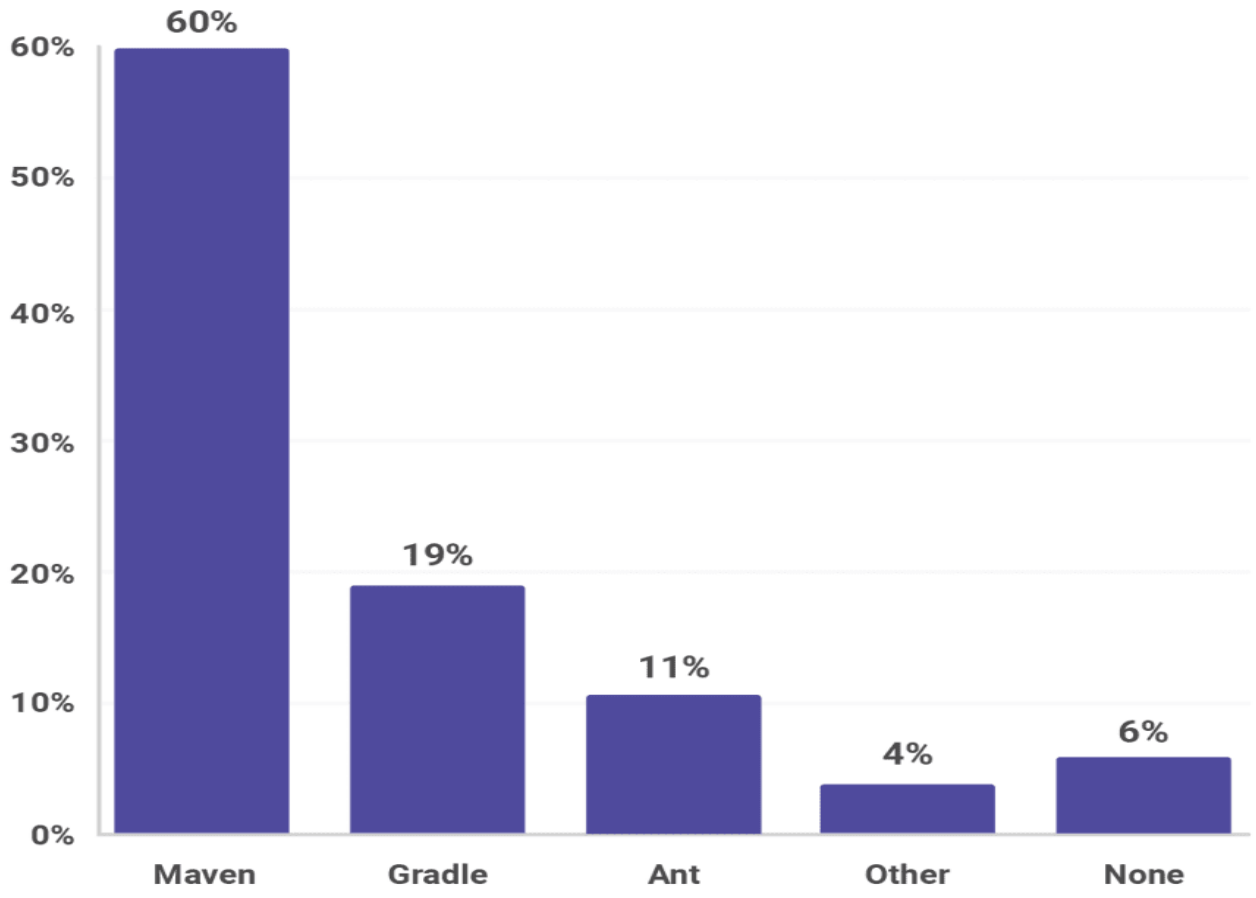
Build systems

Agenda

- Java build systems
- Ant
- Maven
- Gradle

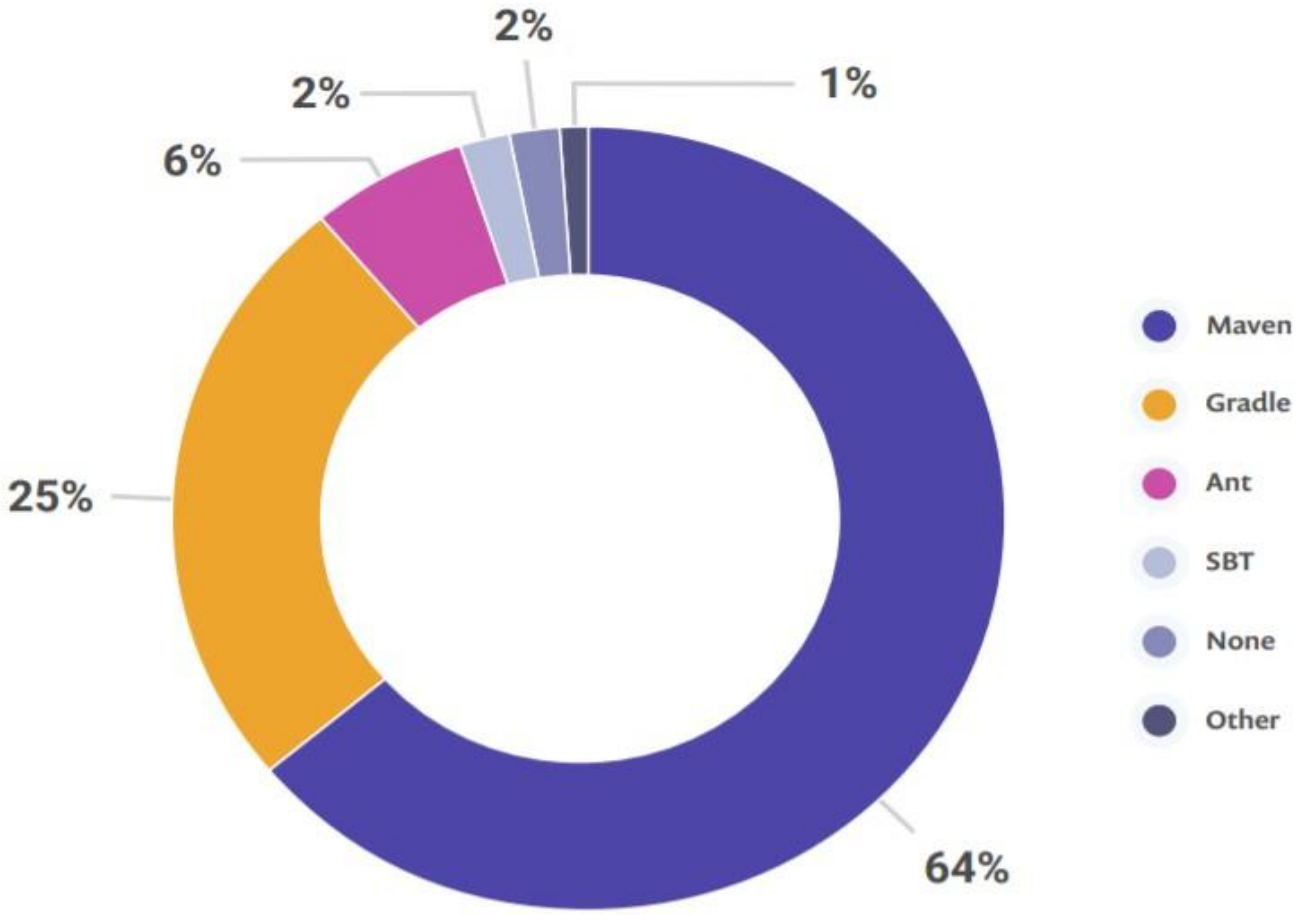
Java build systems

Build systems adoption



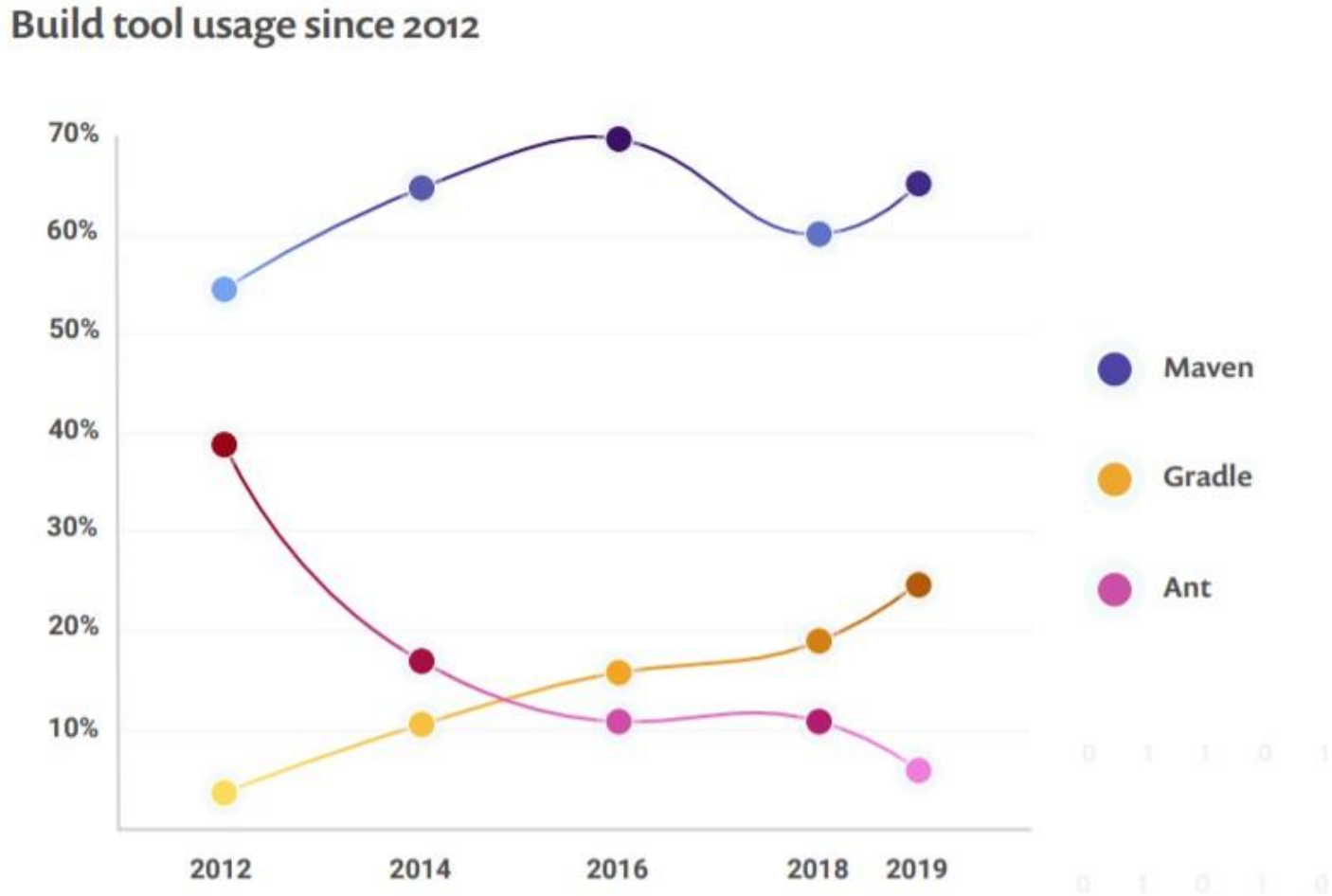
Snyk JVM Ecosystem 2018 report: https://snyk.io/blog/jvm-ecosystem-report-2018-tools/

Build systems adoption



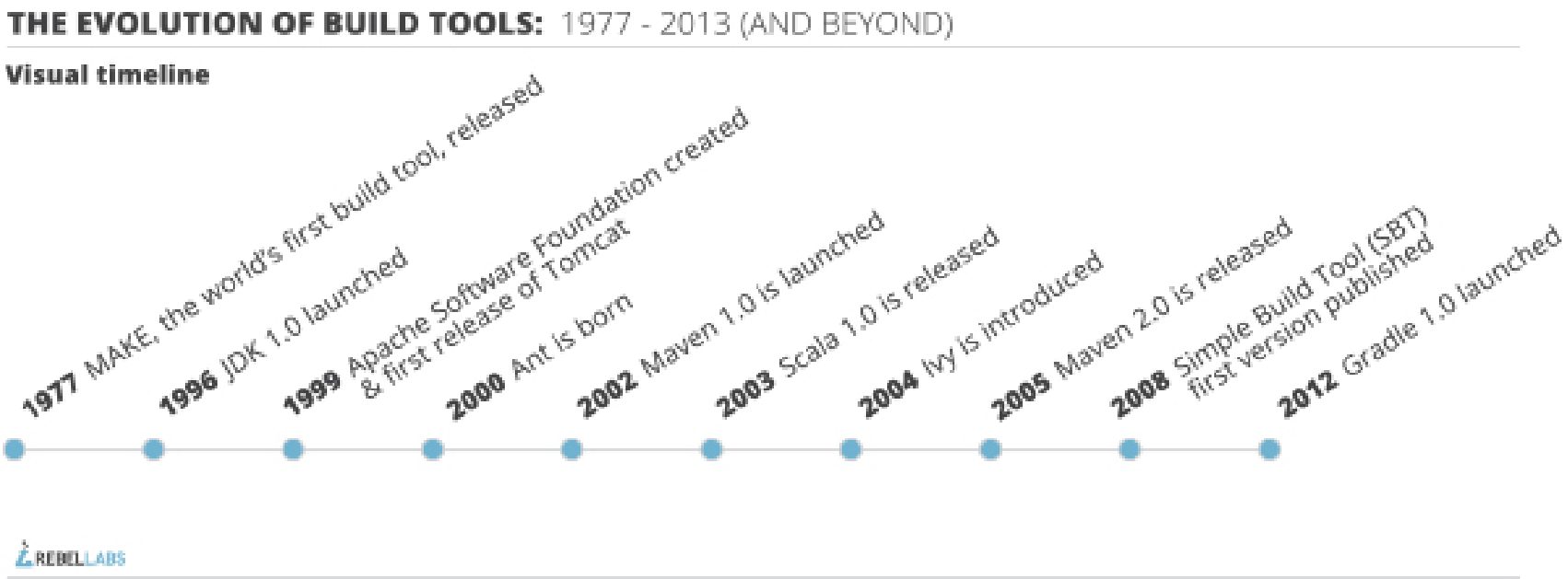
Snyk JVM Ecosystem 2020 report: https://snyk.io/wp-content/uploads/jvm_2020.pdf

Build systems adoption



Snyk JVM Ecosystem 2020 report: https://snyk.io/wp-content/uploads/jvm_2020.pdf

Build systems adoption



https://www.jrebel.com/blog/getting-started-with-java-build-tools

Java build systems

- The three most popular Java build system to date are Ant, Maven and Gradle

- Build systems automate the process of building distribution artifacts of a projects by providing a mechanism for trigger a build process

- A build process can be as simple as:

  - compile the source code with **javac**
  - run the **JUnit** tests
  - build a **JAR** file for the project
  - Copy the JAR file and library JARs to a zip file

Features of a build system

- Build systems provide a common set of features such as:
  - incremental compilation (i.e. compiling only changes sources)
  - management of different profiles (i.e. development and production)
  - managing project versions
  - managing project resources (i.e. properties files, images etc.)

Features of a build system

- Build systems provide a common set of features such as:
  - storing and retrieving third-party libraries in a central repository
  - rich ecosystem of plug-ins
  - a declarative way to describe build steps

Ant

Ant overview

- Ant was created as a replacement of the MAKE build tool
- Written in Java and best suited for Java projects
- Uses XML files to manage the build process
- The default build file used by Ant is called **build.xml**
- A build file contains an Ant project

Ant project

- An Ant project contains one or more Ant targets and properties
- Ant targets define the build logic and may contain one or more Ant tasks (like **javac**, **mkdir** etc.)
- Ant targets may depend on each other thus defining a build order based on a sequence of Ant targets being executed
- Properties are key-value pairs that can be used by targets

Example build.xml

```
<?xml version="1.0"?>
<project name="Hello" default="COMPILE">
       <target name="COMPILE"
       description="compiles Java source code">
         <mkdir dir="classes"/>
         <javac srcdir="." destdir="classes"/>
    </target>
    …
</project>
```

The compile target can executed from the command prompt by navigating to directory of the build.xml file and running:

```
ant COMPILE
```

Ant features

- Built-in features of Ant include:
  - compilation of Java source code (using **javac**)
  - Javadoc generation
  - generation of various archives (such as ZIP, TAR and JAR)
  - Managing files and folders

Ant features

- Built-in features of Ant include:
  - Sending of email notifications
  - Execution of unit tests written in JUnit or TestNG
  - Integration with version control systems such as SVN or Git

Ant extensions

- Ant tasks can also be provided by third-party extensions to Ant
- They are placed as JAR files in the **lib** folder of the Ant installation
- One of the most popular extensions is AntContrib that provides tasks such as:
  - **If**: for conditional execution of other tasks in a target
  - **for**: for looping within a target
  - **switch**: for conditional execution based on a matching value
  - **trycatch**: for try-catch logic of execution
  - **forget**: run sequence of tasks in a separate thread

Maven

Maven overview

- Maven aims to solve some of the limitations of build systems like Apache Ant such as:
  - writing of complex and verbose XML files
  - lack of any imposed project structure
  - lack of built-in dependency management mechanism (although an integration with the Apache Ivy dependency manager is provided)

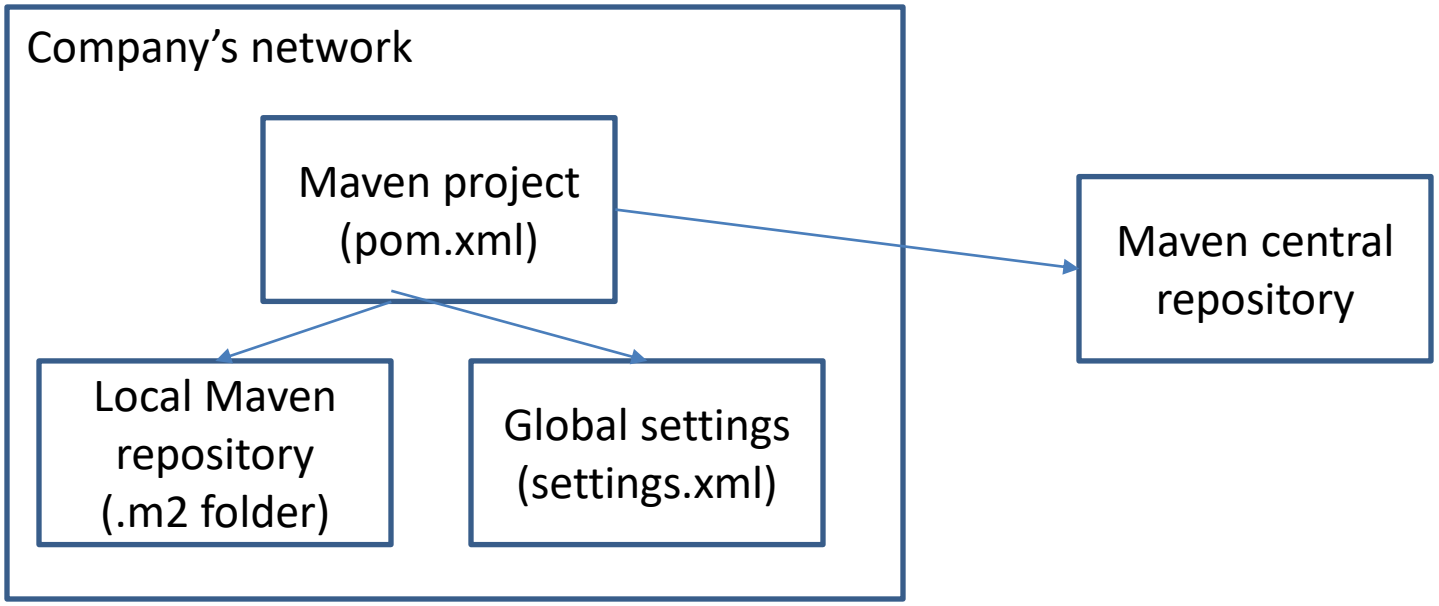
Maven overview

- Maven build files are also based on a XML domain-specific language
- The Maven XML file is less verbose than the one of Ant due to built-in elements for management of dependencies and higher-level plug-ins
- The default build file used by Maven is called **pom.xml**

Maven features

- Main features of Maven include:
  - project dependency management
  - release management and publication
  - multi-module build
  - plug-in system with a number of already existing third-party plug-ins
  - Separation of dependency and plug-in management in the build configuration and Maven repositories
  - possibility to call Ant build targets

Maven architecture



Maven project structure

- Maven enforced a well-defined project structure:
  - **src/main/java**: contains the Java source code of the project
  - **src/main/resources**: contains the resource files used by the project
  - **src/test/java**: contains the unit tests of the project
  - **src/test/resources**: contains the test resource files used by the unit tests
  - **target**: default build output directory, **classes** subfolder contains compiled Java source code

Maven project

- A Maven build file contains a **project** as a start element which may contain a number of elements such as:
  - **groupId**, **artifactId** and **version** of the project
  - parent Maven project
  - name of the project
  - description of the project
  - packaging type of the project (i.e. JAR or POM)

Maven project

- A Maven build file contains a **project** as a start element which may contain a number of elements such as:
  - properties used by the Maven build
  - list of dependencies for the project
  - **build** section with various plug-ins and their configuration used by the build
  - list of dependency/Maven plug-in repositories
  - list of child Maven projects (modules)

Example pom.xml

```
<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
		http://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<groupId>com.martin-toshev</groupId>
	<artifactId>fundamentals</artifactId>
	<version>1.0.0-SNAPSHOT</version>

	<dependencies>
		<dependency>
			<groupId>org.junit.jupiter</groupId>
			<artifactId>
				junit-jupiter-engine</artifactId>
			<version>5.3.1</version>
			<scope>test</scope>
		</dependency>
	</dependencies>
</project>
```

Maven build lifecycle

- Maven works by means of so called **build lifecycle** that defines a sequences of **phases** to be executed

- **Phases** are the Maven equivalent of Ant target whereby each phase may trigger one or more **goals** that perform the actual work

- A **goal** is a unit of work performed by a Maven plug-in that implements the goal

Maven default lifecycle

- Maven provides a default lifecycle that has the following phases:
  - validate
  - generate-sources
  - process-sources
  - generate-resources
  - process-resources
  - compile
  - process-test-sources
  - process-test-resources
  - test-compile
  - test
  - package
  - install
  - deploy

Maven phases

- A maven phase can be executed with the **mvn** tool
- All phases up to the specified one in the default lifecycle are executed in order
- For example the following runs all phases up to the **compile** phase in the default lifecycle:

```
mvn compile
```

Maven dependencies

- Maven dependencies are other Maven projects along with associated build artifacts (such as JAR files)
- Maven dependencies are managed by Maven repositories
- Maven repositories can be public (like Maven central which is the default one is) or private within an organization

Dependency scopes

- Maven dependencies may have a scope associated such as:
  - **compile** (default scope): dependency is required during compilation
  - **provided**: available at compile time but at runtime should be provided by a container
  - **runtime**: dependency is not required for compilation but at runtime
  - **test**: dependency is available for test compilation and execution phases only
  - **system**: JAR file is provided explicitly (i.e. via location on the file system)

Maven multimodule projects

- A multimodule Maven project provides the possibility to build more than one child projects at once in order

```
<modules>
        <module>utilities</module>
        <module>services</module>
        <module>persistence</module>
</modules>
```

Maven profiles

- Maven profiles provide the possibility to create different build logic based on a specified profile (i.e. **development** or **production**)

```
<profiles>
    <profile>
        <id>production</id>
        <build>
            <plugins>
                <plugin>
                //...
                </plugin>
            </plugins>
        </build>
    </profile>
</profiles>
```

```
mvn -P production compile
```

Maven archetypes

- Maven archetypes provide predefined “template” projects that can be generated

- Maven archetypes can be listed (and generated) with the **archetype: generate** goal

```
mvn archetype:generate
```

Maven plug-ins

- Maven provides a rich ecosystem of plug-ins that can be used in Maven builds
- A plug-in provides one or more MOJOs (Maven old Java objects) that implement different plug-in goals
- A plug-in goal can be triggered directly from the command line independently from a build using the following syntax:

```
mvn [plugin-name]:[goal-name]
```

- For example:

```
mvn compiler:compile
```

Maven plug-ins

- Core plug-ins include:
  - **compiler**: for compilation of Java source code
  - **surefire**: for unit test execution
  - **jar**: for building a JAR file from the current project
  - **javadoc**: for Javadoc generation
  - **dependency**: for dependency manipulation
- Full list: https://maven.apache.org/plugins/index.html

Gradle

Gradle overview

- Gradle is a build tool based on the concepts of Ant and Maven
- Gradle build files are based on a Groovy domain-specific language
- The default Gradle build file is called **build.gradle**

Gradle overview

- Gradle enforces the same directory conventions as Maven
- Dependencies are defined using the same format
- Gradle builds may also step on existing Maven repositories
- The **java** Gradle plug-in emulates the Maven default lifecycle

Gradle overview

- A Gradle project consists of one or more Gradle tasks
- Gradle provides a number of predefined tasks
- Properties may be specified in a **gradle.properties** file
- Subprojects (modules) can be specified in a **settings.gradle** file

Example build.gradle

```
apply plugin: 'java'

repositories {
    mavenCentral()
}

jar {
    baseName = 'example'
    version = '1.0.0-SNAPSHOT'
}

dependencies {
    compile 'junit:junit:4.12'
}
```

Example Gradle task

- An Ant task can be called from a Gradle task as follows:

```
task zip {
        doLast {
                ant.zip(destfile: 'archive.zip') {
                        fileset(dir: 'src') {
                                include(name: '**.xml')
                                exclude(name: '**.java')
                        }
                }
        }
}
```

Questions ?