Maven

Taming the Beast

Roberto Cortez

Learning Plan

- Getting started with Maven
- Managing Dependencies
- Plugins and Projects
- Collaboration using Maven

Prerequisites

- JDK 11 or 13 (11 recommended. 8 may work as well)
- Latest version of Maven (3.6.3 or at least 3.6.x)
- Code Editor or IDE (IntelliJ or Eclipse)

Getting started with Maven

Maven Overview

What exactly is Maven?

- Maven is a set of standards, a repository format, and a piece of software used to manage and describe projects

- It defines a standard life cycle for building, testing, and deploying project artifacts

- It provides a framework that enables easy reuse of common build logic for all projects following Maven's standards

A bit of history…

- Maven was born from the desire to make several projects work in the same way
- Standardize on a set of practices
- Stop “reinventing the wheel”

A bit of history…

- Reuse developers knowledge
- Easier for developers to move across projects
- Focus on what needs to be done

Projects benefit from…

- Coherence
- Reusability
- Agility
- Maintainability

Projects benefit from…

- Standardise on a set of best practices
- Effectively reuses the best practices of an entire industry
- It is easier for developers to jump between different projects
- Stop building the build, and start focusing on the application

Getting started with Maven

Maven Principles

Convention over Configuration

- Standard directory layout
- Single Maven project produces a single output
- Standard naming conventions
- Fixed life cycle goals

radcortez / javaee7-angular

Unwatch 49 | Unstar 241 | Fork 216

Code | Issues 2 | Pull requests 0 | Actions | Projects 0 | Wiki | Security | Insights | Settings

Branch: master ▾ javaee7-angular / src / main / | Create new file | Upload files | Find file | History

Roberto Cortez Fixed standalone option to not require NPM build and run out of the box. Latest commit b25427d on Aug 3, 2016

..		
java/com/cortez/samples/javaee7a...	Fixes to be able to run on Glassfish.	6 years ago
resources	Fixed broken images links.	4 years ago
webapp	Fixed standalone option to not require NPM build and run out of the box.	4 years ago

io/smallrye/config/smallrye-config/1.6.0

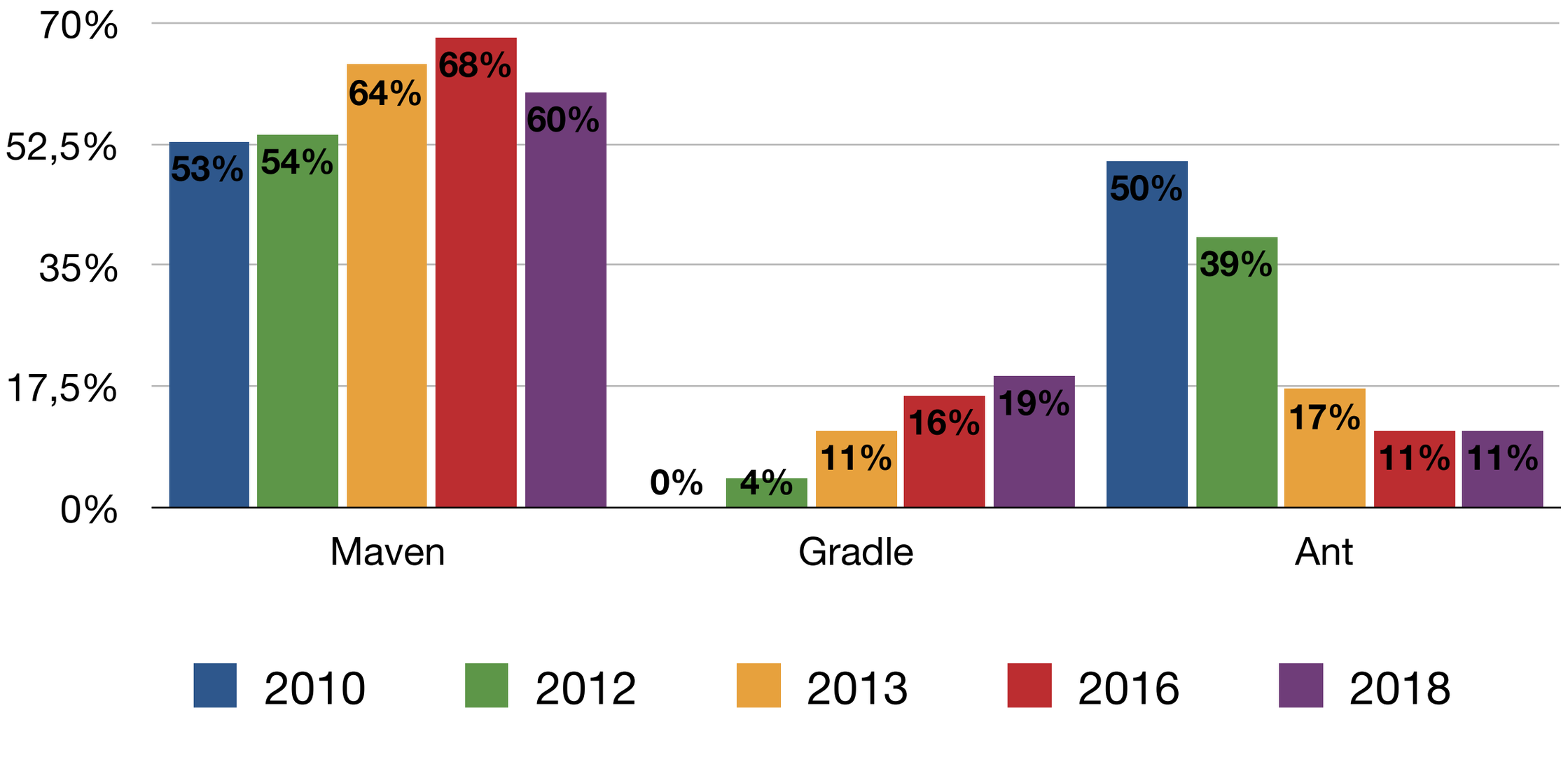
../		
smallrye-config-1.6.0-javadoc.jar	2020-02-06 13:13	114249
smallrye-config-1.6.0-javadoc.jar.asc	2020-02-06 13:13	836
smallrye-config-1.6.0-javadoc.jar.md5	2020-02-06 13:13	32
smallrye-config-1.6.0-javadoc.jar.sha1	2020-02-06 13:13	40
smallrye-config-1.6.0-sources.jar	2020-02-06 13:13	34006
smallrye-config-1.6.0-sources.jar.asc	2020-02-06 13:13	836
smallrye-config-1.6.0-sources.jar.md5	2020-02-06 13:13	32
smallrye-config-1.6.0-sources.jar.sha1	2020-02-06 13:13	40
smallrye-config-1.6.0.jar	2020-02-06 13:13	71711
smallrye-config-1.6.0.jar.asc	2020-02-06 13:13	836
smallrye-config-1.6.0.jar.md5	2020-02-06 13:13	32
smallrye-config-1.6.0.jar.sha1	2020-02-06 13:13	40
smallrye-config-1.6.0.pom	2020-02-06 13:13	3392
smallrye-config-1.6.0.pom.asc	2020-02-06 13:13	836
smallrye-config-1.6.0.pom.md5	2020-02-06 13:13	32
smallrye-config-1.6.0.pom.sha1	2020-02-06 13:13	40

Benefits

- You don't have to worry about whether or not it's going to work
- You don't have to jump through hoops trying to get it to work
- It should rarely, if ever, be a part of your thought process

Like the engine in your car or the processor in your laptop, a useful technology just works, in the background, shielding you from complexity and allowing you to focus on your specific task

Popularity


70%
52,5%
35%
17,5%
0%
53%
54%
64%
68%
60%
0%
4%
11%
16%
19%
50%
39%
17%
11%
11%
Maven
Gradle
Ant
2010
2012
2013
2016
2018

Facts

- It’s around for more than 15 years
- Is not going away anytime soon
- It became the de-facto standard for building Java projects
- Embrace it

Getting started with Maven

Using Maven

The POM

- Everything in Maven is driven in a declarative fashion using Maven's Project Object Model (POM)

- The plugin configurations contained in the POM describe your build

- The POM is an XML document held in a file named pom.xml in the root of your project

The POM

```xml
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.mycompany.app</groupId>
    <artifactId>my-app</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

The POM

- The POM will allow you to compile, test, and generate basic documentation

- How is this possible using a 15 line file?

The Super POM

- Maven's Super POM is the analog of the Java language's java.lang.Object class.
- All POMs have an implicit parent which is Maven's Super POM.
- Super POM contains important default information so
- You don't have to repeat this information in the POMs you create.

The Build Lifecycle

- Software projects generally follow a similar build path: preparation, compilation, testing, packaging, installation.

- The build life cycle consists of a series of phases where each phase can perform one or more actions

- In Maven, you do work by invoking particular phases in this standard build life cycle.

The Build Lifecycle

- The standard build life cycle consists of many phases and these can be thought of as extension points.

- When you need to add some functionality to the build life cycle you do so with a plugin.

- Maven plugins provide reusable build logic that can be slotted into the standard build life cycle.

Build Lifecycle Phases

- validate
- compile
- test
- package
- verify
- install
- deploy

Dependency Management

```
<dependency>
    <groupId>javax.persistence</groupId>
    <artifactId>persistence-api</artifactId>
    <version>1.0</version>
    <scope>provided</scope>
</dependency>
```

Dependency Management

- Where does that dependency come from?

- Where is the JAR?

Dependency Management

- A dependency is a reference to a specific artifact that resides in a repository
- A dependency is uniquely identified by the following identifiers: groupId, artifactId and version
- Optionally they can include a type and classifier
- Maven dependencies are declarative

“At a basic level, we can describe the process of dependency management as Maven reaching out into the world, grabbing a dependency, and providing this dependency to your software project.”

Repositories

- A Repository is just an abstract storage mechanism
- In practice the repository is a directory structure in your file system
- Maven has two types of repositories: local and remote

Local Repository

- Your local repository is one-stop-shop for all artifacts that you need
- It works as a cache for all your projects
- You only download each artifact once

Remote Repository

- Remote repositories refer to any other type of repository
- Protocols such as file:// and http://.
- These repositories are set up by a third party to provide their artifacts for downloading.
- Maven Central is the main remote repository

Repositories

- Maven tries to satisfy each dependency by looking in all of the remote repositories
- If a matching artifact is located, Maven transports it to your local repository
- Every project with a POM that references the same dependency will use this single copy

Recap

- Convention over Configuration
- Build Life cycle
- Dependency Management
- Repositories

My First Maven Project

mvn archetype:generate

QA

Break Time!

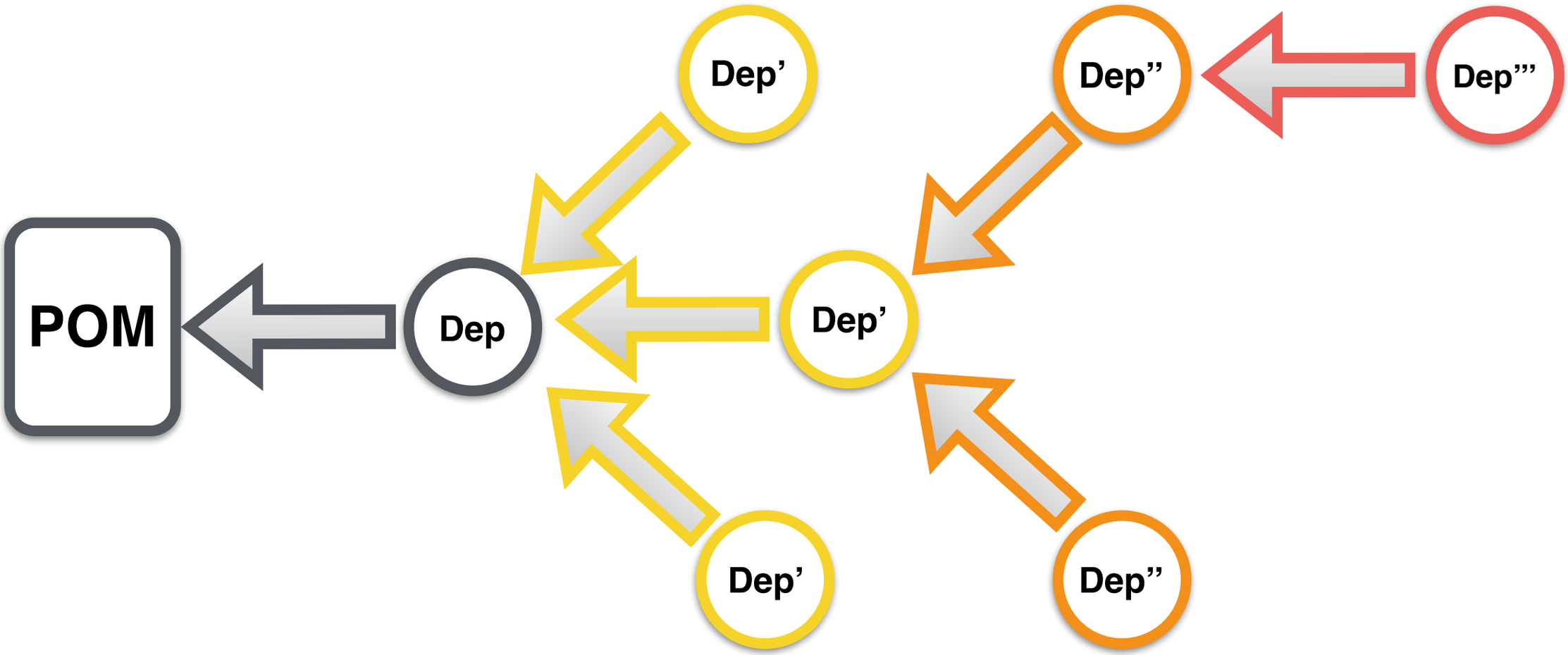
Dependencies

Managing Maven Dependencies

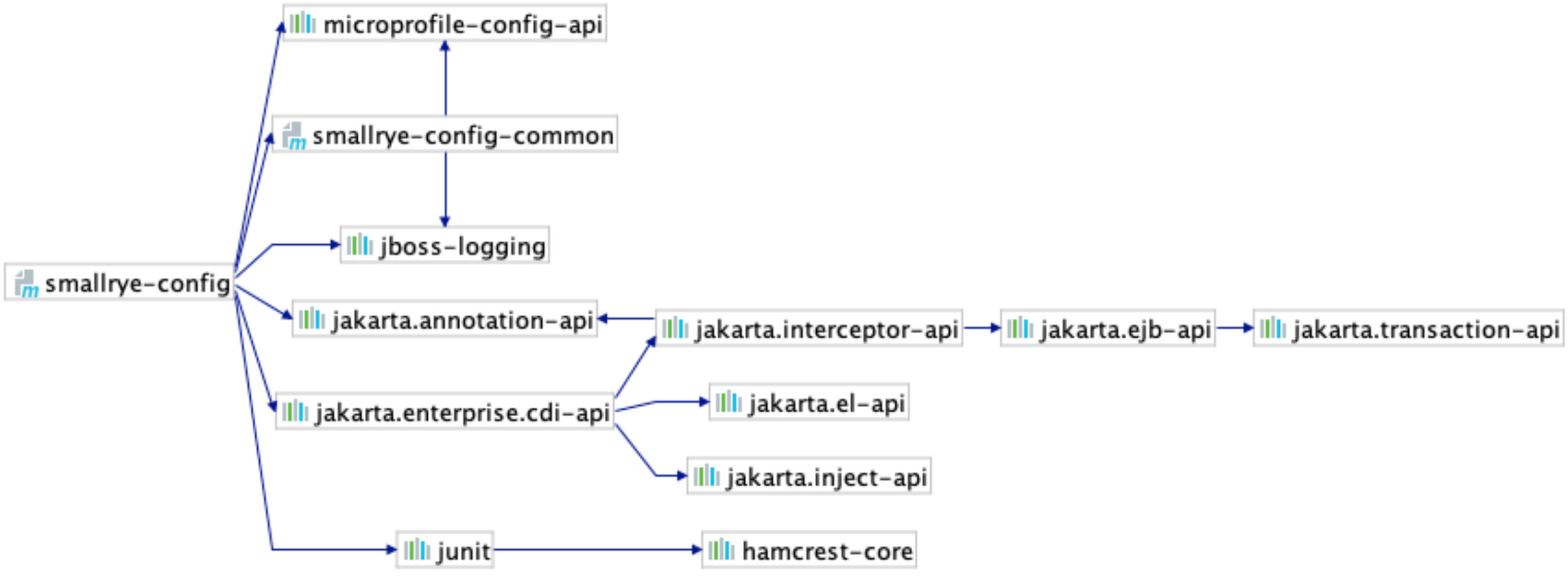
Transitive Dependencies

- Discover libraries that your own dependencies require
- No limit to the number of levels
- The graph of included libraries can quickly grow quite large
- May cause a problems if a cyclic dependency is discovered

Why is this JAR in my build?




0% of the Internet downloaded
Saving:
theinternet.zip from the
Estimated time left: 4,3
Download to: C:\
Transfer rate: 41.
Close this dialog box when download completes
Open
Open Folder
Cancel
Error Downloading the Internet
Insufficient memory for drive C:\ .
Insert disk in drive A:\
OK


microprofile-config-api
smallrye-config-common
jboss-logging
smallrye-config
jakarta.annotation-api
jakarta.interceptor-api
jakarta.ejb-api
jakarta.transaction-api
jakarta.enterprise.cdi-api
jakarta.el-api
jakarta.inject-api
junit
hamcrest-core

Why is this JAR in my build?

- Include ONLY what you need!
- Use **mvn dependency:analyze**
- Add used dependencies
- Some trial and error involved

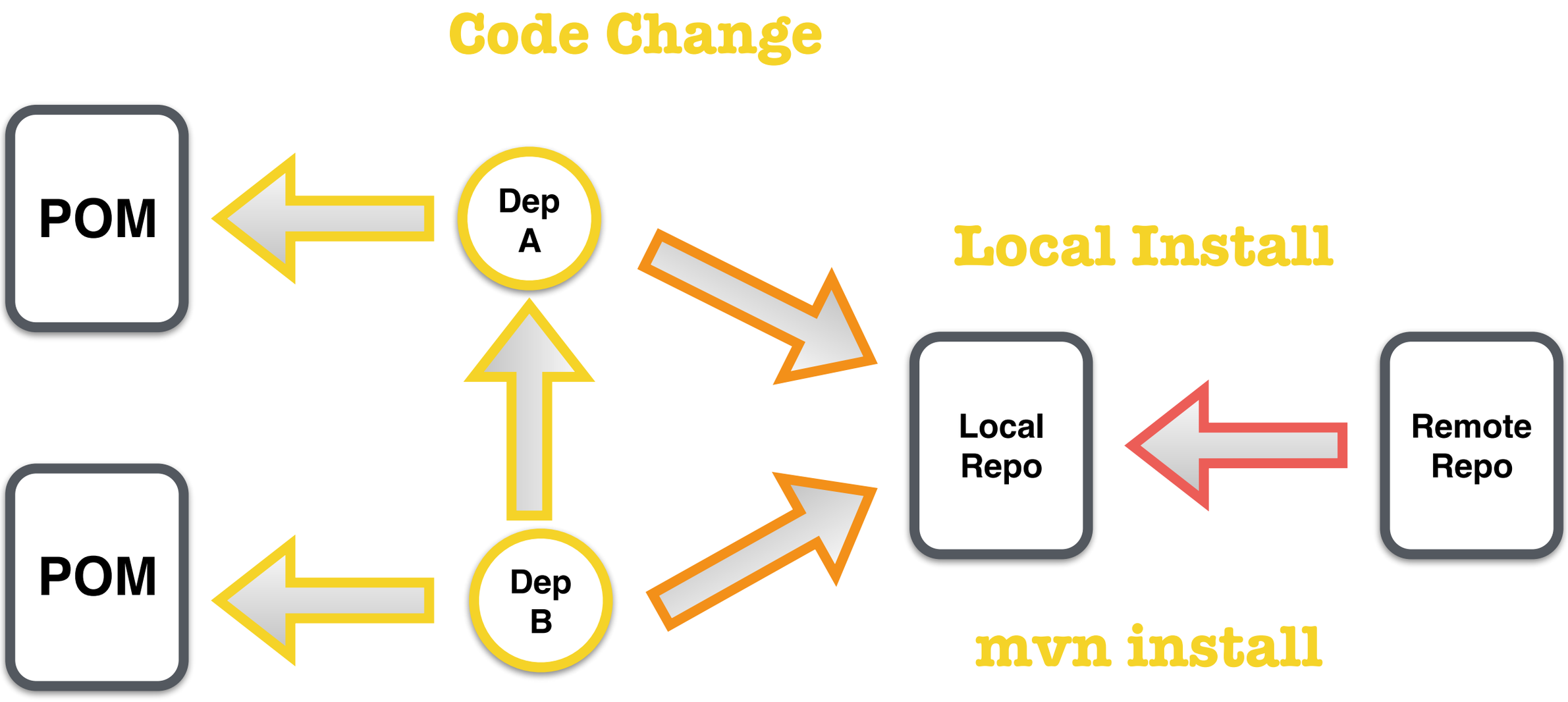
I can’t see my changes

- Dependencies are not build in the local repo
- Dependency version is not correct
- Remote repo has overwritten changes

Remember Repositories?

- Local and Remote
- Checks for Remote repo for dependency
- Downloads to Local and use if from there from that point forward

I can’t see my changes



I can’t see my changes

- That’s why one of the first things you learn is **mvn install**!
- Places the artifact in your local repository
- All the other projects can use it now

Dependency Scopes

- Used to calculate the various classpaths used for compilation, testing, and so on
- It also assists in determining which artifacts to include in a distribution of the project
- Each of the scopes affects transitive dependencies in different ways.

Dependency Scopes

- Compile
- Runtime
- Provided
- Test
- System
- Import

Optional Dependencies

- The optional flag
- Lets other projects know that you do not require this dependency in order to work correctly.
- If C is an optional dependency of B, C will not be included in the artifact resolution for project A.

Dependencies Exclusions

- Unwanted dependencies included in your project
- Exclude specific dependencies
- On exclusion the artifact will not be added to your project

```xml
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>6.14.3</version>
    <scope>compile</scope>
    <exclusions>
        <exclusion>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

A JAR is not being included

- Usually related with Dependency Scopes
- Might also be excluded somewhere
- May cause ClassNotFoundException

A JAR is not being included

- Use **mvn dependency:list**
- Use **mvn dependency:tree**
- Change dependencies to use the appropriate scope

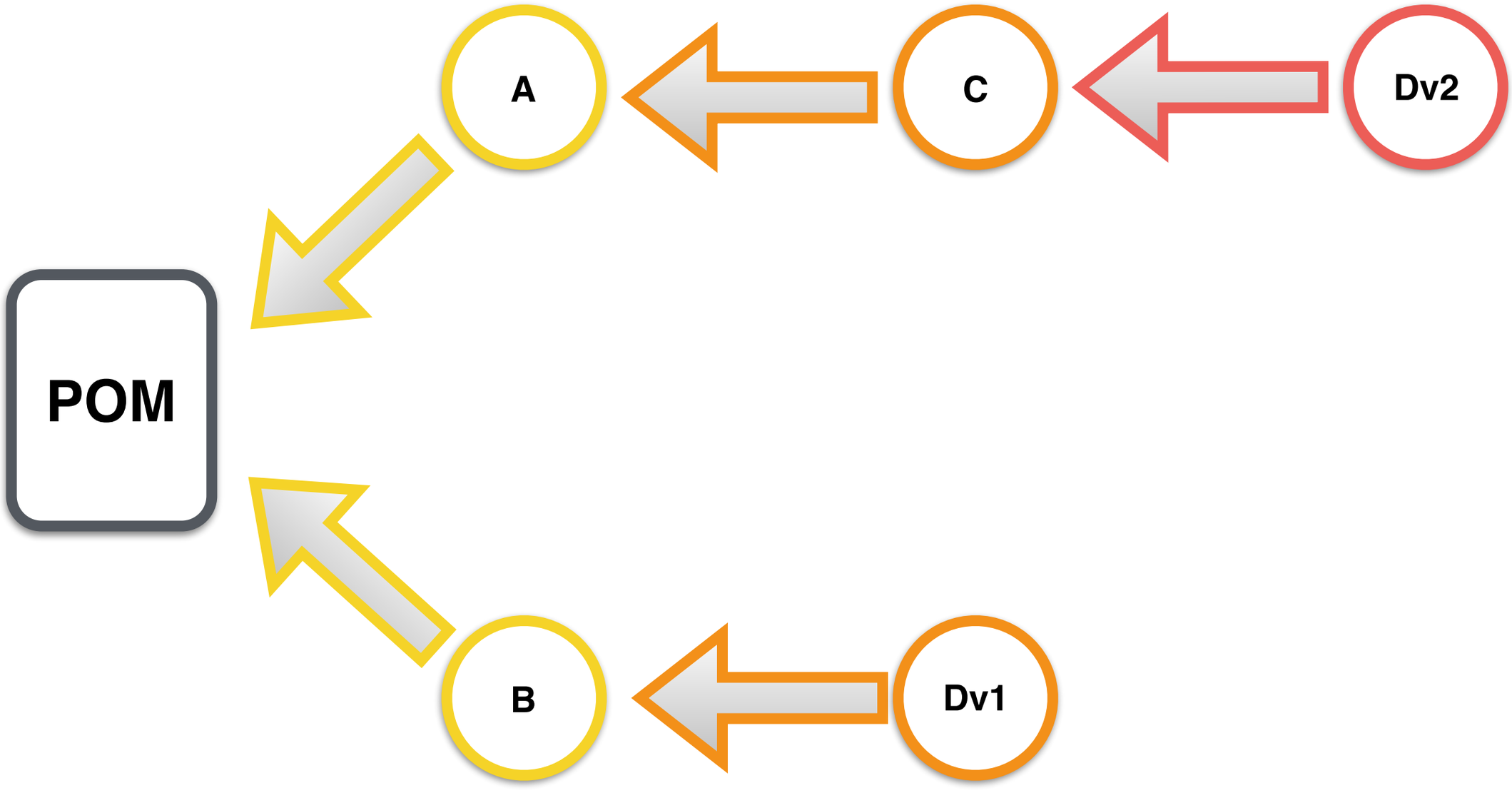
Version Ranges

- x.y.z indicate the preferred dependency version to use
- Maven has no knowledge about what versions will work
- In the case of a conflict, Maven uses the “nearest dependency” technique.
- You can force Maven to use a version that fit in a given range.

Version Ranges

Range	Meaning
(,x.y]	Less than or equal to x.y
[x.y,x.z]	Between x.y and x.z (inclusive)
[x.y,z.y)	Greater than or equal to x.y, but less than z.y
[x.y,)	Greater than or equal to x.y
(,x.y),(x.y,)	Any version, except x.y

I can’t see my changes!



Who wins?

Dv1

Dependency Mediation

Dependency Mediation

- Nearest Definition: means that the version used will be the closest one to your project in the tree of dependencies.

- If two dependency versions are at the same depth in the dependency tree, it's the order in the declaration that counts.

- You could explicitly add a dependency to D1 in A to force the use of D1… or just exclude D2.

I can’t see my changes!

1.0.2c

1.0.2

1.0.2a

1.0.02

Which one is the Latest version?

1.0.2.a

Versions

- Order by well-known qualifiers and lexically
- Check ComparableVersion
- https://cwiki.apache.org/confluence/display/MAVENOLD/Versioning

Snapshot Versions

- A SNAPSHOT version for a dependency means that Maven will look for new versions
- SNAPSHOT dependencies are assumed to be changing, so Maven will attempt to update them
- When you specify a non-snapshot version of a dependency Maven will download that dependency once and never attempt to retrieve it again

Snapshot Versions

- SNAPSHOT version may break your build

- When releasing your project you should get rid of all SNAPSHOTs

Dependency Snapshots

- Snapshots were designed to be used in a team environment
- Share development versions of artifacts that have already been built.
- Build involves checking out all of the dependent projects and building them yourself.
- You must build all of the modules simultaneously from a master build.

Dependency Snapshots

- It relies on manual updates from developers
- There is no common baseline against which to measure progress
- Building can be slower as multiple dependencies must be rebuilt also
- Changes developed against outdated code can make integration more difficult

Classifiers

- The classifier allows to distinguish artifacts that were built from the same POM but differ in their content
- Displayed as groupId:artifactId:packaging:classifier:version
- Consider a project that targets different JRE versions
- Attach secondary artifacts, like source code, API docs, etc

Classifiers

- While not recommended, you can solve design flaws using classifiers
- It’s against to the separation of concerns (SoC) principle, so you should avoid it
- Nevertheless, it could help you with generating client artifacts (ejb-client)

Managing Dependencies

- Many dependencies to manage
- Number of dependencies only increases over time
- Combine the power of project inheritance with specific dependency management elements in the POM
- Manage, or align, versions of dependencies across several projects

Managing Dependencies

- The dependencyManagement element is used only to state the preference for a version and does not affect a project's dependency graph

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>javax.persistence</groupId>
            <artifactId>persistence-api</artifactId>
            <version>1.0</version>
            <scope>provided</scope>
        </dependency>
```

Managing Dependencies

- The version for this dependency is derived from the dependencyManagement element which is inherited from the top-level POM.

```
<dependencies>
    <dependency>
        <groupId>javax.persistence</groupId>
        <artifactId>persistence-api</artifactId>
        <scope>compile</scope>
    </dependency>
```

I can’t find the Artifact!

- “Could not resolve dependencies… Could not find artifact”

- Even worst than the Java NPE

I can’t find the Artifact!

- Dependency defined correctly?
- Remote repo contains the artifact?
- Most recent pom version?
- Jar is corrupted?
- Cached artifact?
- Dependency overriden?

I can’t find the Artifact!

- Artifacts get renamed all the time
- The artifact might be in another remote repo
- POM parent might not be the latest

Debug Commands

- mvn help:effective-pom
- mvn help:effective-settings
- mvn -Dmaven.repo.local=/temp/.m2
- mvn versions:display-dependency-updates

Recap

- Transitive Dependencies
- Dependency Scopes
- Snapshot Dependencies
- Managing Dependencies

Dependencies

Add Dependencies to your Maven Project

QA

Break Time!

Plugins and Projects

Plugins

- Maven is actually a platform that executes plugins within the build life cycle

- Tasks of the build process are executed by the set of plugins associated with the phases of a project's build life-cycle.

- Extends a project's build to incorporate new functionality, such as integration with external tools and systems.

Plugins in the Life Cycle

Lyfe-cycle Phase	Goal	Plugin
process-resources	resources	maven-resources-plugin
compile	compile	maven-compiler-plugin
process-test-resources	testResources	maven-resources-plugin
test-compile	testCompile	maven-compiler-plugin
test	test	maven-surefire-plugin
package	jar	maven-jar-plugin
install	install	maven-install-plugin
deploy	deploy	maven-deploy-plugin

Plugins

- The plugins provided “out of the box” by Maven are enough to satisfy the needs of most build processes
- It is likely that a plugin already exists to perform a particular task you are looking for
- You may need to write a custom plugin to integrate into the build life cycle

Plugins

- Build plugins are executed during the build and configured in the build section

- Reporting plugins are executed during the site generation and configured in the reporting section

```xml
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-source-plugin</artifactId>
      <version>3.0.1</version>
      <executions>
        <execution>
          <id>attach-sources</id>
          <goals>
            <goal>jar</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Plugins

- You can configure plugins with parameters (java versions, paths, behaviour, etc)

- You can attach a plugin execution to a particular phase of the build lifecycle

- You can also set up a pluginManagement section that works like the dependencyManagement to describe general plugin configuration

Project Inheritance

- State your organizational, deployment or common dependencies in a single place

- POMs that lets you draw on the resources stated in the specified top-level POM

- You can inherit down to any level you wish - enabling you to add resources where it makes sense in the hierarchy of your projects.

Aggregation

- A project with modules is known as a multimodule, or aggregator project

- Maven will topologically sort the modules such that dependencies are always build before dependent modules

- Inheritance and aggregation create a nice dynamic to control builds through a single, high-level POM

Multiple Modules

- A POM represent a single module
- You require a POM file in the root of the project
- You can replicate the project structure as sub modules
- The root POM becomes the parent POM

Multiple Modules

- Aggregator modules do not have any folders (source or resources)
- Their POM has to be of a special packaging called “pom”
- Each subfolder goes into a module section
- Each of the modules has to respect Maven’s structure

```
<groupId>org.eclipse.microprofile.config</groupId>
<artifactId>microprofile-config-parent</artifactId>
<version>2.0-SNAPSHOT</version>
<packaging>pom</packaging>
<name>MicroProfile Config</name>
<description>Eclipse MicroProfile Config Feature :: Parent POM</description>
<url>http://microprofile.io</url>

<modules>
    <module>api</module>
    <module>tck</module>
    <module>spec</module>
</modules>
```

Inheritance vs Aggregation

- These are two different things
- A POM may be inherited from but does not necessarily aggregate any modules
- Both are POM projects

Profiles

- Profiles are Maven's way of letting you create variations in the build life cycle to build on different platforms or build with different JVMs, etc

- Profiles modify the POM at build time, and are meant to be used in complementary sets to give equivalent-but-different parameters to each build

Profiles

- Per Project - Defined in the POM itself
- Per User - Defined in the Maven-settings
- Global - Defined in the global maven-settings
- Profile descriptor - a descriptor located in project basedir (unsupported in Maven 3)

Profiles

- Can be specified explicitly using a command line option
- Can be activated in the Maven settings
- Can be triggered automatically based on the detected state of the build environment

```
<profiles>
  <profile>
    <id>quality-checks</id>
    <activation>
      <property>
        <name>skipChecks</name>
        <value>!true</value>
      </property>
    </activation>
    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-checkstyle-plugin</artifactId>
          <executions>
            <execution>
              <id>verify-style</id>
              <phase>process-test-classes</phase>
              <goals>
                <goal>check</goal>
              </goals>
            </execution>
          </executions>
          <configuration...>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```

Profiles can modify…

- dependencies, dependencyManagement
- plugins, pluginManagement
- properties, modules, reporting, repositories

Properties

- Values placeholders
- Accessible anywhere in the POM
- Uses the notation ${propertyName}

Properties

- Maven already sets a bunch of properties that you can use
- Useful to set particular variables in a single place and then reuse across yours projects
- Can come from environment variable, settings, java system properties

```xml
<properties>
  <java.version>11</java.version>
  <maven.compiler.target>${java.version}</maven.compiler.target>
  <maven.compiler.source>${java.version}</maven.compiler.source>

  <plugin.version.geronimo.openapi>1.0.11</plugin.version.geronimo.openapi>
  <plugin.version.frontend>1.7.6</plugin.version.frontend>
  <plugin.version.talkdesk.swagger>1.0</plugin.version.talkdesk.swagger>

  <dependency.version.lombok>1.18.10</dependency.version.lombok>
  <dependency.version.quarkus>999-SNAPSHOT</dependency.version.quarkus>
  <dependency.version.mapstruct>1.3.1.Final</dependency.version.mapstruct>

  <dependency.version.flyway.junit>1.0</dependency.version.flyway.junit>
</properties>
```

Recap

- Plugins
- Inheritance and Aggregation
- Profiles
- Properties

Plugins and Projects

Enhance your Maven Project

QA

Break Time!

Collaborating using Maven

Project Health

- Through the POM, Maven has access to the information that makes up a project

- With tooling, Maven can analyze, relate, and display that information in a single place

- When referring to health, there are two aspects to consider

Project Health

- Code quality - determining how well the code works, how well it is tested, and how well it adapts to change

- Project vitality - finding out whether there is any activity on the project, and what the nature of that activity is

Project Health

- Accuracy – whether the code does what it is expected to do

- Robustness – whether the code gracefully handles exceptional conditions

- Extensibility – how easily the code can be changed without affectingor requiring changes to a large amount of other code.

- Readability – how easily the code can be understood

Project Health

- Maven takes all of the information under the project Web site.
- Generate reports that provide information about the project.
- Each report is suited to monitor a specific concept of the project.
- Choosing which reports to include, and which checks to perform during the build determine the effectiveness of your build reports.

Project Health

Report	Description	Notes
Javadoc	Produces an API from Javadoc	Useful for most Java software. Important for any projects publishing a public API
Checstyle	Checks your source code against a standard descriptor for formatting issues	Use to enforce a standard code style. Recommended to enhance readability of the code
JaCoCo	Analyze code statement coverage during unit tests or other code execution	Recommended for teams with a focus on tests. Can help identify untested or even unused code
Surefire	Show the results of unit tests visually	Recommended for easier browsing of test results. Shows any tests that are long and slowing the build.

Project Site

- Additional reports go into the reporting section
- You can also add the team members, mailing lists, issue tracking, license information, etc.
- Use mvn site

```xml
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-javadoc-plugin</artifactId>
      <version>3.1.1</version>
    </plugin>
  </plugins>
</reporting>
```

Team Collaboration

- Software development, whether it is 2 or 200 people, faces a number of challenges
- This problem gets exponentially larger as the size of the team increases.
- As teams continue to grow, disseminate all of the available information about a project is nearly impossible

Team Collaboration

- Encompass a set of practices and tools that enable effective team communication and collaboration

- Provides teams with real-time information on the builds and health of a project, through the practice of continuous integration

- Setup a consistent developer environment

Continuos Integration

- Continuous integration enables automated builds of your project
- Ensures that conflicts are detected earlier
- CI can enable a better development culture
- Team members can make smaller, iterative changes

“Continuous integration is a key element of effective collaboration”

Continuos Integration

- Commit early, commit often.
- Run builds as often as possible.
- Fix builds as soon as possible.
- Establish a stable environment.
- Run clean builds.
- Run comprehensive tests.
- Build all of a project's active branches.
- Run a copy of the application continuously.

Improve your Build

- Use a Plugin and Dependency Management section
- Specify versions
- Avoid SNAPSHOTS
- Use Global Properties
- Use Modules

Speedup your Build

- By default, modules are built sequentially
- Depending on your project, you can run the build in parallel
- It can speed your build substantially

Speedup your Build

- mvn -T 2 install (2 Threads)

- mvn -T 1C install (1 Thread per core)

Speedup your Build

- Build only what you need
- Skip tests
- Offline
- Tune JVM options

Speedup your Build

- mvn install -pl moduleName -am / -amd
- mvn -Dmaven.test.skip=true
- -XX:+TieredCompilation -XX:TieredStopAtLevel=1
- -XX:+TieredCompilation -XX:TieredStopAtLevel=1
- mvn -o

Final Considerations

- Everyone should be responsible
- Keep it clean
- If you feel that you are fighting with the tool, you are probably doing it wrong

Recap

- Project Health
- Team Collaboration
- Continuous Integration (and Deployment)
- Improve your Build

“Never trust the IDE!

If it works on the command-line then the IDE is the problem! ”

QA

Thank you!