

Software Design and Construction
159.251
Process Automation

Amjed Tahir
a.tahir@massey.ac.nz

Summary

- Agile methodologies and the need for build automation
- Build Tools - overview
- ANT
- Maven
- Gradle
- NPM

Short Iterations

- Software projects run in iterations
- At the end of an iteration, working (tested) code is produced
- This code can be used by the user - good for checking that the programme is actually fulfilling customer requirements

Enabling Short Iterations

- Each iteration includes a set of activities with an approximately constant (and significant) cost C :
 - compile
 - package
 - test
 - document
 - deliver
- Having many (N) short iterations is expensive: $N \cdot C$

Automate these activities, reduce overhead (C) to close to zero

Build Tools

Build Tools

Build tools were created to automate much of the process of taking source code and creating a fully functional programme.

Build tools are responsible for:

- Sourcing dependencies
- Compiling source code
- Running tests
- Packaging bytecode up into executables (e.g. JARs)
- Creating reports

There are many other things that can be put into build tools!

When to Build

- **On demand:** Build is triggered explicitly by the programmer - We focus on this in this topic.
- **Triggered:** Build starts when a certain event happens such as code being committed to GitHub - This is Continuous Integration (covered later in the course).
- **Scheduled:** Built periodically (e.g. nightly builds)

Build Tool Systems

Most ecosystems have their own build tools.

- Javascript - NPM
- Python - Pip
- Ruby - Rake
- .NET - NuGet
- PHP - Packagist
- Perl - CPAN
- JVM - Ant, Maven, Gradle, sbt (Scala)

We'll focus on the Java tools, along with a brief look at NPM. The principles of how build tools work all are reasonably similar.

Build Tool Syntax

Build tools come in three main flavours:

1. They use the syntax of their programming language (e.g. Rake uses Ruby syntax)
2. They use general-purpose markup languages (e.g. XML for Ant or Maven, JSON for NPM)
3. They use domain specific languages (DSL) to make their own custom syntax (e.g. Gradle uses Groovy)

Build Tool History

- Early programmes were commonly compiled using command line
- When these became more complicated, shell scripts could be used
- In the 70s, MAKE was created which abstracted some of the shell commands and automated some shell tasks
- Since then, each build tool has added different layers of abstraction on top of these original OS commands

ANT

The first Java build tool

ANT

- Java programmers initially used MAKE but it didn't work that well for Java (it was intended for C)
- Around 2000, ANT was spawned from the Apache Tomcat project (Tomcat is a web server)
- It is flexible, but ultimately requires complex configuration to do basic tasks, so was superseded by Maven

So why discuss it?

- It is still used in legacy applications a lot
- It is good for demonstrating the basics of what build tools do - we will build on these ideas

ANT Scripts

- ANT operations are defined in the **build.xml** file
- The root element is called **project**
- The next level consists of **targets and properties**
- Targets describe tasks that have to be performed (compile, test, jar, ...)
- Targets are implemented using **tasks**
- Tasks are reusable modules to build scripts
- ANT has a large number of **predefined tasks** on board, but it is also possible to get and install additional tasks, or to write your own tasks (using Java)

A Simple ANT build.xml

```
<project name="HelloWorld" basedir="." default="compile">
  <property name="src.dir" value="src"/>
  <property name="build.dir" value="build"/>
  <property name="classes.dir"
            value="${build.dir}/classes" />

  <target name="compile">
    <mkdir dir="${classes.dir}"/>
    <javac srcdir="${src.dir}"
           destdir="${classes.dir}"/>
  </target>
</project>
```

Targets and Dependencies

- Targets are the main steps in the workflow
- Targets may **depend on** other targets
- If a target is executed, the targets it depends on must be executed first
- A project has a **default target**
- You can run ant in the terminal:

```
ant [build-script]
```

- ANT can also be executed on a particular target:

```
ant [build-script] target
```

Dependencies

- The main limitation of ANT is in managing dependencies.
- Whenever we use code outside of the standard libraries (i.e., dependencies) we need to download those.
- Originally ANT had no functionality for this - you would have to have it ready on your file system.
- Modern tools solved this issue and gained popularity quickly (more on this later_
- ANT later added in IVY, a plugin that deals with dependency management.

Maven

Convention over Configuration

Maven

Maven is the successor build tool managed by the Apache foundation (like ANT):

- It was built to reduce the size and complexity of build scripts, which is a huge issue with ANT scripts, and build in dependency management into the system
- Maven focuses on two aspects:
 - convention over configuration
 - symbolic dependencies with repository-based dependency resolution
- Note that many of the ideas presented in ANT above are available in Maven or are happening behind the scenes

Core concepts

- The **POM** is an xml file (**pom.xml**) that has the project configuration
- Maven creates **artefacts** - usually jar files of executable code
- Artefacts have a composite name consisting of group id, artefact name and a version
- An artefact may **depend on** other artefacts
- Artefacts are resolved over the network against a central **Maven repository** when maven runs, e.g. <https://mvnrepository.com/>
- The repository is searchable to locate artefacts

Maven forces developers to use the standard work cycle. If you want to use a custom lifecycle, the customisation process can be very difficult or even impossible. This allows Maven to deliver a standard package easily.

Example pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<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>nz.ac.massey.sdc</groupId>
  <artifactId>taxcalculator</artifactId>
  <version>1.0-SNAPSHOT</version>

</project>
```

What's interesting is what **isn't** here. This pom.xml can do the entire software lifecycle - compiling, testing, creating an executable JAR etc are all possible with this little information. Compare that with ANT!

Phases

- Maven builds are executed in **lifecycle phase**
- There are dependencies between phases
- Syntax (from terminal): **mvn test**
- Build outputs are stored by default in a **/target** folder
Maven generates

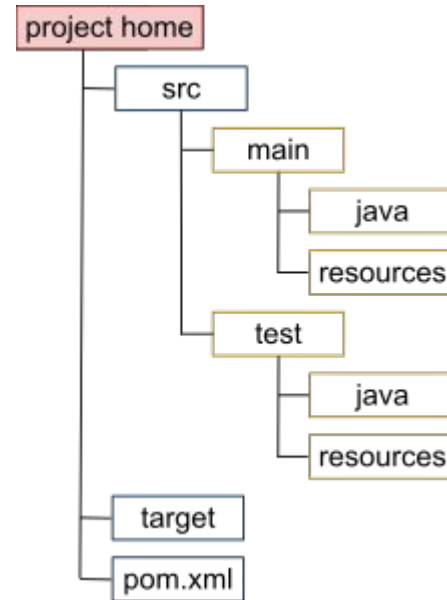
Standard Phases (Selection):

1. **validate**: validate the project is correct and all necessary information is available
2. **compile**: compile the source code of the project
3. **test**: test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
4. **package**: take the compiled code and package it in its distributable format, such as a JAR.
5. **verify**: run any checks to verify the package is valid and meets quality criteria
6. **install**: install the package into the local repository, for use as a dependency in other projects locally
7. **deploy**: done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.
8. **clean**: cleans up artifacts created by prior builds
9. **site**: generates site documentation for this project

Convention over Configuration

- It is extremely simple to start a basic Maven project
- When files are put in standard locations and follow standard naming conventions, “everything works”
- In particular, Java projects should use the **standard Maven project layout**
- Tests (junit classes) have to follow certain naming patterns to be recognised as tests (for instance ***Test**, ***Tests**, or **Test***)
- all conventions can be overruled with configurations (in `pom.xml`)

Test Names



This is a sample Maven file structure. Note that Maven follows the test naming convention: class `Foo.java` test should be called `FooTest.java` (i.e., adding `Test` to the end of the class name). If you named it `FooT.java`, Maven would not find the tests.

The Maven Project Layout

src/main/java	Application/Library sources
src/main/resources	Application/Library resources
src/main/filters	Resource filter files
src/main/assembly	Assembly descriptors
src/main/config	Configuration files
src/main/scripts	Application/Library scripts
src/main/webapp	Web application sources
src/test/java	Test sources
src/test/resources	Test resources
src/test/filters	Test resource filter files
src/site	Site
LICENSE.txt	Project's license
NOTICE.txt	Notices and attributions ..
README.txt	Project's readme

Customising Maven

- Conventions can be overridden and redefined using archetypes
- An archetype is essentially a project template
- Plugins can be used to further customise Maven, in particular to add build functionality (phases) or custom reporting
- Plugins are registered in the `<plugins>` sections of the pom

Dependency Resolution

- Maven manage dependencies automatically
- Instead of copying libraries into the project, it is sufficient to declare them (using group+name+version)
- References are resolved against a repository
- Maven will fetch the respective artefacts during a build, and further artefacts they might depend on
- This means that an initial build can take a long time, but Maven will try to cache artefacts locally

pom.xml – Project using JUnit 5

```
1<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3  <modelVersion>4.0.0</modelVersion>
4
5  <groupId>lectures251</groupId>
6  <artifactId>testing</artifactId>
7  <version>0.0.1</version>
8  <packaging>jar</packaging>
9
10 <name>testing</name>
11 <url>http://maven.apache.org</url>
12
13<properties>
14  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15  <maven.compiler.target>1.8</maven.compiler.target>
16  <maven.compiler.source>1.8</maven.compiler.source>
17</properties>
18
19<dependencies>
20<dependency>
21  <groupId>org.junit.jupiter</groupId>
22  <artifactId>junit-jupiter</artifactId>
23  <version>5.5.0</version>
24  <scope>test</scope>
25</dependency>
26</dependencies>
27
28<build>
29  <plugins>
30    <plugin>
31      <artifactId>maven-jar-plugin</artifactId>
32      <version>2.4</version>
33    </plugin>
34    <plugin>
35      <groupId>org.apache.maven.plugins</groupId>
36      <artifactId>maven-surefire-plugin</artifactId>
37      <version>2.22.1</version>
38    </plugin>
39  </plugins>
40</build>
41
42</project>
43
```

Maven Integration

- All major IDEs have built-in Maven support, or plugins providing it
- Usually, Maven is a project type that can be selected when creating new projects.
- Maven then takes over classpath / buildpath management
- Note that Maven needs an internet connection (in order to download dependencies from the central Maven repository)

Pause: Maven Repository

Maven has built in support in IDEs (Eclipse, IntelliJ, VScode), so you can have your pom.xml automatically created for you.

But what about fetching dependencies? Let's go look at how that works:

<https://mvnrepository.com/>

Using Maven via Terminal

Ensure that you have set the OS path variable to Maven's bin folder. `mvn --version` will check this is set correctly.

Run the following command (adjusting as needed):

```
mvn -B archetype:generate
-DarchetypeGroupId=org.apache.maven.archetypes
-DgroupId=com.mycompany.app -DartifactId=my-app
```

This will generate a folder called `my-app` with a package called `com.mycompany.app`, and importantly, the `pom.xml`.

Depending on your Java version (check `java -version`), you will need to add the following to your pom file (specifying the Java version in your environment - here it is version 16 or higher).

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.2</version>
      <configuration>
        <source>16</source>
        <target>16</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```


Using Maven via Terminal (ctd)

In the folder where the pom.xml file is located (this should be the project's root folder), you can run any of the standard phases.

For example:

```
mvn compile (does the equivalent of javac)
```

```
mvn test (additionally runs tests)
```

```
mvn package (additionally creates jar)
```

These will all compile the project. To run the compiled project, you can use the following command (or run the jar):

```
mvn exec:java
```

```
"-Dexec.mainClass=com.mycompany.app.App"
```

This is equivalent to `cd target/classes/` **and running**

```
java com/mycompany/app/App
```

Using Maven via Eclipse

- To create a new Maven project, press `File`, `new...`, `other`. Maven has a project category there.
- This will generate the `pom.xml`
- Right-click on the `pom.xml` or the project in the package explorer. Under `Run As...` the main Maven lifecycle tasks will be available.

Gradle

A build tool without the XML

Gradle

Gradle was released by Google in 2012, who made it the default for Android application.

It builds on the functionality of Ant and Maven.

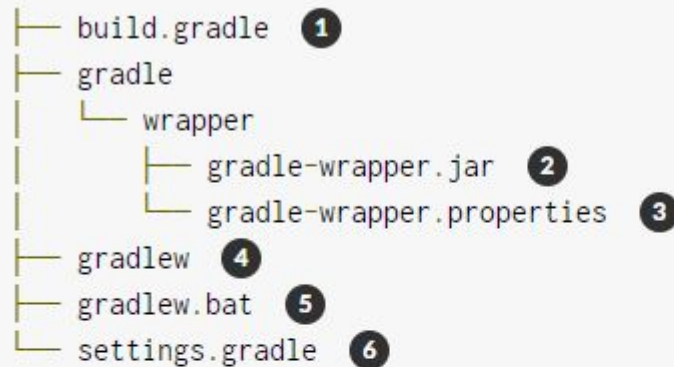
- Uses a DSL (domain specific language) called Groovy
- Is plugin based - has little functionality of its own
- Initially used Ivy for dependency resolution but now uses an in-house solution.
- Uses many of the tasks from Maven by default
- Uses the same default file structure as Maven (but is easy to change)

build.gradle

- Gradle centres around the `build.gradle` file.
- Like Ant and Maven, Gradle can have multiple build files linked together in larger projects.
- The `build.gradle` file uses Groovy or Kotlin syntax
- `settings.gradle` is used for specifying additional information

gradle init command

Groovy



- ❶ Gradle build script for configuring the current project
- ❷ [Gradle Wrapper](#) executable JAR
- ❸ Gradle Wrapper configuration properties
- ❹ Gradle Wrapper script for Unix-based systems
- ❺ Gradle Wrapper script for Windows
- ❻ Gradle settings script for configuring the Gradle build

What gets generated for you: <https://guides.gradle.org/creating-new-gradle-builds/>

Eclipse autogenerated build.gradle

```
build.gradle  settings.gradle

1 /*
2  * This file was generated by the Gradle 'init' task.
3  *
4  * This generated file contains a sample Java Library project to get you started.
5  * For more details take a look at the Java Libraries chapter in the Gradle
6  * user guide available at https://docs.gradle.org/4.8.1/userguide/java\_library\_plugin.html
7  */
8
9 plugins {
10     // Apply the java-library plugin to add support for Java Library
11     id 'java-library'
12 }
13
14 dependencies {
15     // This dependency is exported to consumers, that is to say found on their compile classpath.
16     api 'org.apache.commons:commons-math3:3.6.1'
17
18     // This dependency is used internally, and not exposed to consumers on their own compile classpath.
19     implementation 'com.google.guava:guava:23.0'
20
21     // Use JUnit test framework
22     testImplementation 'junit:junit:4.12'
23 }
24
25 // In this section you declare where to find the dependencies of your project
26 repositories {
27     // Use jcenter for resolving your dependencies.
28     // You can declare any Maven/Ivy/file repository here.
29     jcenter()
30 }
31
```

Adjusted for JUnit5



```
build.gradle  settings.gradle  Library.java  LibraryTest.java

1 plugins {
2     id 'java-library'
3 }
4
5 dependencies {
6     testImplementation 'org.junit.jupiter:junit-jupiter-api:5.5.0'
7     testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.5.0'
8 }
9
10 repositories {
11     mavenCentral()
12 }
13
14 test {
15     useJUnitPlatform()
16 }
17
18 jar {
19     manifest.attributes "Main-Class": "Library"
20 }
21
```


Notes

- There are several predefined interfaces in Gradle such as plugins, dependencies etc. These take **closures** which are curly braces with some properties defined or methods to run.
- Dependencies are only used at specific points in the project lifecycle. Examples:
 - implementation - compile-time dependency
 - runtimeOnly - run-time dependency
 - testImplementation - for compile time tests

Gradle on the command line

Once installed, the following commands are useful:

- `gradle init`
starts a project in your cwd
- `gradle tasks`
lists the tasks available given your plugins
- `gradle build`
compiles and tests your programme

Note that changes are tracked - if you haven't updated source code, the tasks might end up being skipped.

NPM

The Node.js package manager

NPM

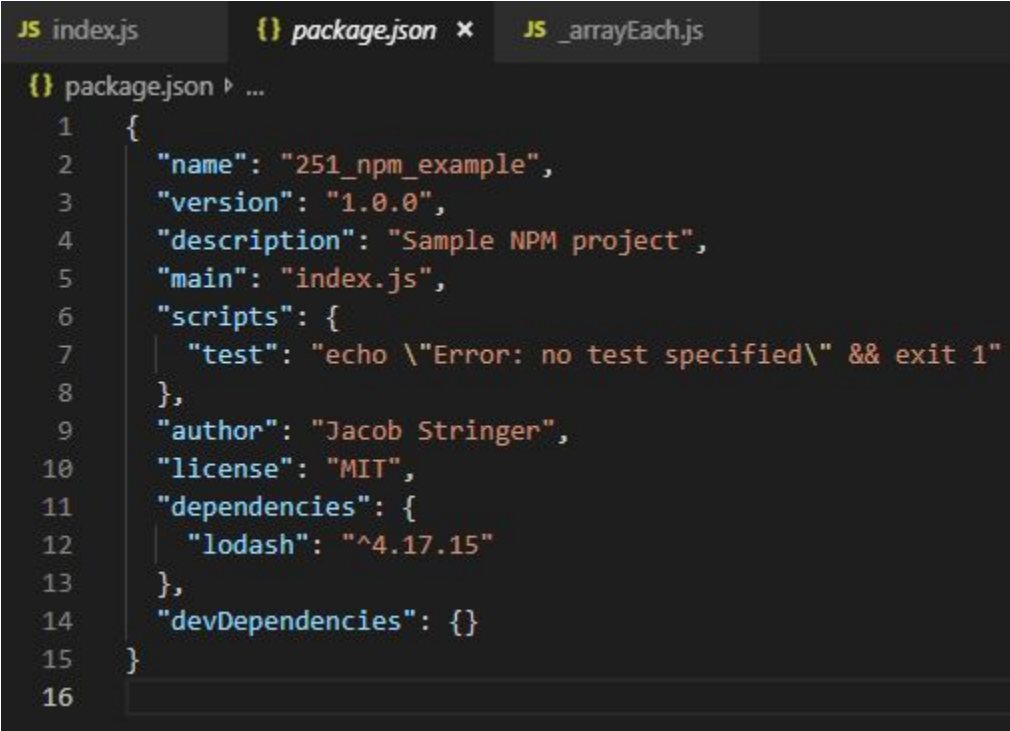
Node Package Manager is widely used through the JS ecosystem. It requires node.js to be installed to use.

- Revolves around the `package.json` manifest file
- Distinguishes between runtime and development dependencies (similar to the compile time, run time distinction in Maven and Gradle)
- It will be quite interesting to discuss when we discuss package versions

A good intro video for those interested:

<https://www.youtube.com/watch?v=jHDhaSSKmB0>

Sample package.json file



The image shows a code editor with three tabs: 'index.js', 'package.json', and '_arrayEach.js'. The 'package.json' tab is active, displaying a JSON object with the following properties: 'name' (251_npm_example), 'version' (1.0.0), 'description' (Sample NPM project), 'main' (index.js), 'scripts' (test: echo \"Error: no test specified\" && exit 1), 'author' (Jacob Stringer), 'license' (MIT), 'dependencies' (lodash: ^4.17.15), and 'devDependencies' (empty object). The code is syntax-highlighted and includes line numbers from 1 to 16.

```
1  {
2    "name": "251_npm_example",
3    "version": "1.0.0",
4    "description": "Sample NPM project",
5    "main": "index.js",
6    "scripts": {
7      "test": "echo \"Error: no test specified\" && exit 1"
8    },
9    "author": "Jacob Stringer",
10   "license": "MIT",
11   "dependencies": {
12     "lodash": "^4.17.15"
13   },
14   "devDependencies": {}
15 }
16
```