Build and Dependency Management

Isaac Griffith

Fall 2021

Idaho State University | Software Engineering

SE 5520 - Software Construction and Configuration Management

Outcomes

At the end of Today's Lecture you will be able to:

- Understand why we use build and dependency management tools
- Understand the basics of gradle
- Initiate a gradle project
- Utilize the basic gradle tasks to build a java project
- Configure a gradle project

Inspiration

Build Management

SE 5520

Build Tools?

Build Tools

- Provide the capability to manage and automate the build process

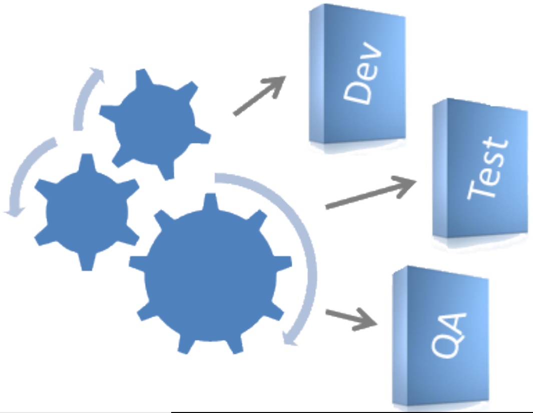
Dependency Management

- Provides the capability to automate the management of third-party libraries

Software What?

Building Software Sucks!

- Software Build Process
  - Develop
  - Test
  - Assemble
  - Deploy
  - Integrate
  - Repeat (again and again and again)



SE 5520

What is Gradle?

- Gradle is a general purpose **build system**
- It comes with a rich build description language (DSL) based on **Groovy**
- It supports "**build-by-convention**" principle
- But it is very **flexible** and **extensible**
- It has **built-in plug-ins** for Java, Groovy, Scala, Web, OSGi
- It derives all the best and integrates well with **Ivy, Ant and Maven**

What is Gradle?

- Gradle is also a dependency management system
- It downloads required libraries (with specific versions) for use in your project.
- Gradle is similar to other tools used in other languages
  - Python has `pip`
  - JavaScript has `npm`
  - C# has `nuget`
  - C++ has `cmake` and `conan`
  - Ruby has `bundler`

Gradle Features

- Declarative builds and build-by-convention
- Language for dependency based programming and many ways to manage dependencies
- Groovy as a base langauge allows imperative programming

Gradle Features

- Deep and rich API for managing projects, tasks, dependency artifacts and much more
- State of the art support for multi-project builds
- Ease of integration and migration
- Free and open source

Advanced Features

- Parallel unit test execution
- Dependency build
- Incremental build support
- Dynamic tasks and task rules
- Gradle daemon

Using Gradle

SE 5520

A Java Project

build.gradle file

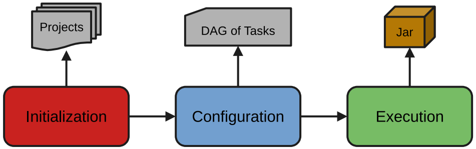
```
plugins {
  id 'java'
}

repositories {
  mavenCentral()
}

dependencies {
  testRuntime "org.junit.jupiter:junit-jupiter-engine:5.5.2"
  testRuntime "org.junit.platform:junit-platform-runner:1.5.2"
}

test {
  useJUnitPlatform()
}
```

Core Concepts

- **Build Script**: a build configuration script supporting one or more project
- **Project**: a component that needs to be built. It is made up of one or more tasks
- **Task**: a distinct step required to perform the build. Each task/step is atomic (either succeeds or fails).
- **Publication**: the artifact produced by the build process

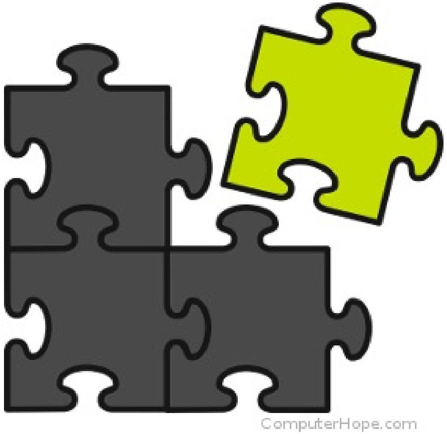
Dependency Resolution

- **Dependencies**: tasks and projects depending on each other (internal) or on third-party artifacts (external).
- **Transitive dependencies**: the dependencies of a project may themselves have dependencies
- **Repositories**: the “places” that hold external dependencies (Maven/Ivy repos, local folders).
- **DAG**: the directed acyclic graph of dependencies (what depends on what)
- **Dependency configurations**: named sets (groups) of dependencies (e.g. per task)



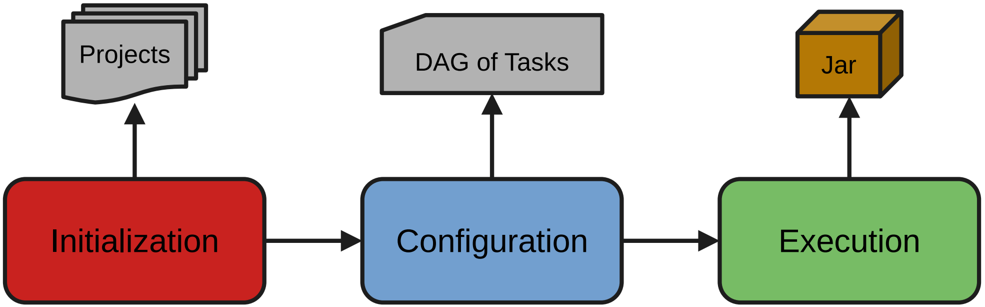
Plugins

- A plugin applies a set of extensions to the build process.
  - Add tasks to a project
  - Pre-configure these tasks with reasonable defaults.
  - Add dependency configurations
  - Add new properties and methods to existing objects
- Plugins implement the "build-by-convention" principle in a flexible way



The Build Lifecycle

1. **Initialization**: initialization of the project
2. **Configuration**: configuration of the project (computes the DAG)
3. **Execution**: executes the sequence of build tasks



A Simple Example

SE 5520

Initiating a Project

- To initilize a project as a gradle project, you need to:
  - include a "build.gradle" in the root project directory
  - setup the proper directory structure
- Alternatively, you can let gradle do this for you by
  - Executing the following in the root project directory

```
> gradle init
```

Run a build task

```
> gradle test
```

Compiles the source and runs the tests

```
> gradle tasks
```

clean, assemble, build, classes, testClasses, test, jar, etc

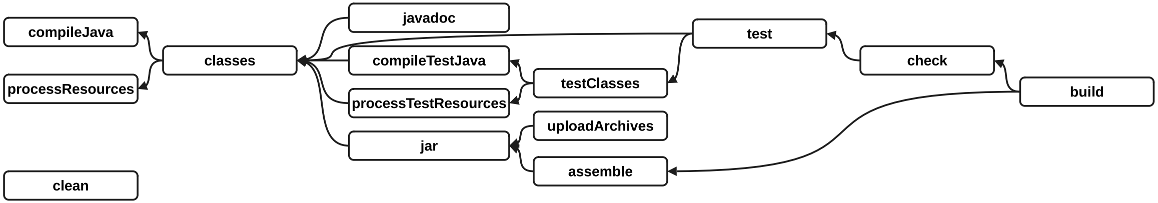
Standard Java Tasks

Tasks added by Java Plugin

- `compileJava`
- `jar`
- `javadoc`
- `clean`
- `test`

Lifecycle Tasks

- `assemble`
- `check`
- `build`



Another Example

SE 5520

Dependency Management

SE 5520

Repository Configuration

```
repositories {
  mavenCentral()
}
```

```
{
  mavenCentral name: 'single-jar-repo', urls: "http://repo.mycompany.com/jars"
  flatDir name: 'localRepository',
  dirs: 'lib' flatDir dirs: ['lib1', 'lib2']
}
```

Referencing Dependencies

```
dependencies {
  testImplementation 'junit:junit:4.7'
  implementation group: 'org.springframework', name: 'spring-core', version: '2.5'
}
```

- General Syntax
  - `<configuration> '<reference-string>'`
  - `<configuration> group: '<group-name>', name: '<artifact-name>', version: '<version>'`

Dependency Configurations

- Plugins like `java` and `groovy` have predefined dependency configurations, but you may also create your own

```
configurations {
  foobar
}

dependencies {
  foobar 'junit:junit:4.7'
}
```

Built-in Java Configurations

- `implementation` - implementation only dependencies
  - extends `compile`
- `compileOnly` - compile time only dependencies, not used at runtime
- `compileClasspath` - compile classpath, used when compiling source. Used by task `compileJava`
  - extends `compile`, `compileOnly`, `implementation`
- `annotationProcessor` - annotation processors used during compilation
- `runtimeOnly` - runtime only dependencies
- `runtimeClasspath` - runtime classpath contains elements of the implementation, as well as runtime only elements
  - extends `runtimeOnly`, `runtime`, `implementation`

Built-in Java Configurations

- `testImplementation` - implementation only dependencies for tests
  - extends `testCompile`, `implementation`
- `testCompileOnly` - additional dependencies only for compiling tests, not used at runtime
- `testCompileClasspath` - test compile classpath, used when compiling test sources. Used by task `compileTestJava`
  - extends `testCompile`, `testCompileOnly`, `testImplementation`
- `testRuntimeOnly` - runtime only dependencies for running tests
  - extends `runtimeOnly`
- `testRuntimeClasspath` - runtime classpath for tunning tests. Used by task `test`
  - extends `testRuntimeOnly`, `testRuntim`, `testImplementation`
- `archives` - artifacts (e.g., jars) produced by this project. Used by task `uploadArchives`

Using Gradle Plugins

SE 5520

Extending Your Build

Any Gradle script can be a plug-in:

```
apply from: 'otherScript.gradle'
apply from: 'http://mycomp.com/otherScript.gradle'
```

Use many of the standard or 3rd-party plug-ins:

```
plugins {
  id 'java'
  id 'groovy'
  id 'scala'
  id 'war'
}
```

Standard Plugins

Plug-in ID	Plug-in ID
base	application (java, groovy)
java-base	jetty (war)
groovy-base	maven (java, war)
scala-base	osgi (java-base, java)
reporting-base	war (java)
java (java-base)	code-quality (reporting-base, java, groovy)
groovy (java, groovy-base)	eclipse (java, groovy, scala, war)
scala (java, scala-base)	idea (java)
antlr (java)	project-report (reporting-base)
announce	sonar
java-library	jacoco
spotbugs	pmd

Resources

- Getting Started Guide
- Plugin Reference
- Plugin Development Tutorials

Summary

For Next Time

Are there any questions?