INTRODUCTION TO GRADLE

GABOR BATA

MAY 18, 2015

AGENDA

1. Gradle overview
2. Getting started
3. Projects, tasks
4. Plugins
5. Java plugin
6. Behind the scenes
7. Summary

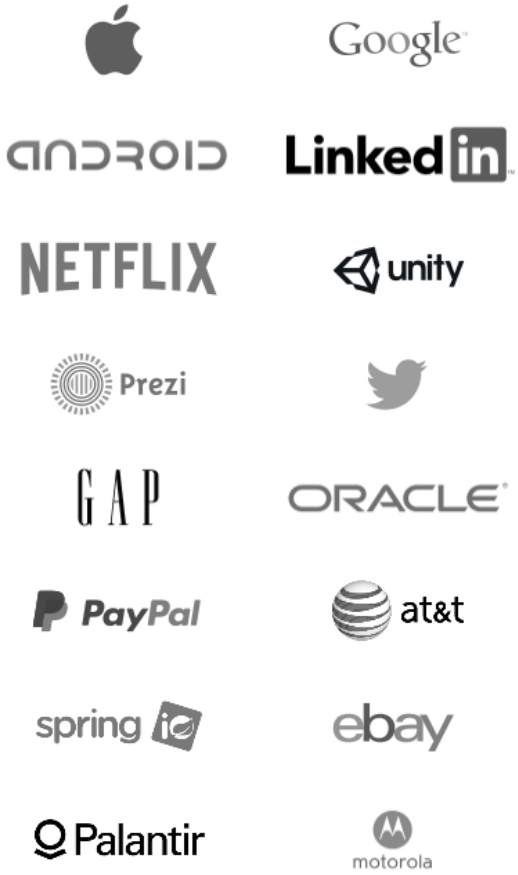
GRADLE OVERVIEW

Gradle is an open source build automation system which can automate the building, testing, publishing, deployment and more of software packages or other types of projects such as generated static websites, generated documentation or anything else.

Features

- Combines the power and flexibility of Ant with
- The dependency management and conventions of Maven
- Instead of XML, it had its own clear and compact DSL, based on Groovy

Do I need to know Groovy?
Not necessarily, unless you really want to stray away from convention and do things your own way.
On the other hand, knowing Groovy will help understanding what happens behind the scenes.



GETTING STARTED

1. Download Gradle from **gradle.org**
2. Create an environment variable `GRADLE_HOME` and point it to the Gradle Installation folder
3. Add `%GRADLE_HOME%\bin` to the PATH environment variable

After these, we can configure our Gradle build by using the following configuration files:

- Build script `build.gradle` specifies a project and its tasks.
- Properties file `gradle.properties` is used to configure the properties of the build (optional).
- settings file `settings.gradle` is optional in a build which has only one project. With more projects, it describes which projects participate to our build.

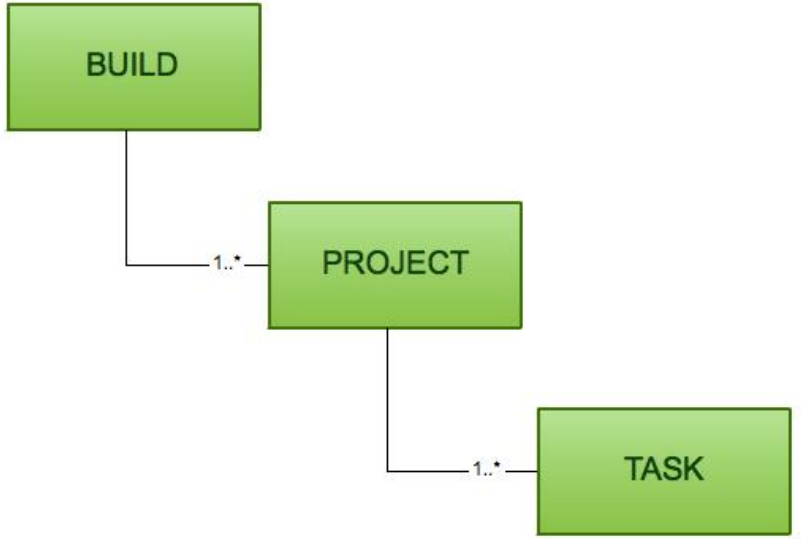
You can also generate build.gradle with the `gradle init` command. If it is called in a Maven folder then Gradle tries to convert `pom.xml` to a Gradle build script. This is currently in *incubating* phase and works only for simpler projects.

PROJECTS, TASKS

Gradle has two basic concepts: projects and tasks.

- A **project** is either something we build (e.g. a jar file) or do (deploy our application to production environment). A project consists of one or more tasks.
- A **task** is an atomic unit work which is performed our build (e.g. compiling our project or running tests).

The relationships between these concepts are illustrated in the following figure:



PROJECTS, TASKS - DEFINE TASKS

BUILD.GRADLE

```
task build << {
  println 'Building the project...'
}

task hello << {
  println 'hello'.capitalize()
}

task world(dependsOn: hello) << {
  println 'world'.capitalize()
}

build.dependsOn world

defaultTasks 'build'
```

OUTPUT

```
$ gradle
:hello
Hello
:world
World
:build
Building the project...

BUILD SUCCESSFUL
```

PROJECTS, TASKS - TASK TYPES

There are many built-in task types, e.g.:

- Checkstyle - Runs Checkstyle against some source files.
- Delete - Deletes files or directories.
- Exec - Executes a command line process.
- Jar - Assembles a JAR archive.
- Zip - Assembles a ZIP archive.
- etc.

Built-in copy task, e.g:

```
task copyDocs(type: Copy) {
  from 'src/main/doc'
  into 'build/target/doc'
}
```

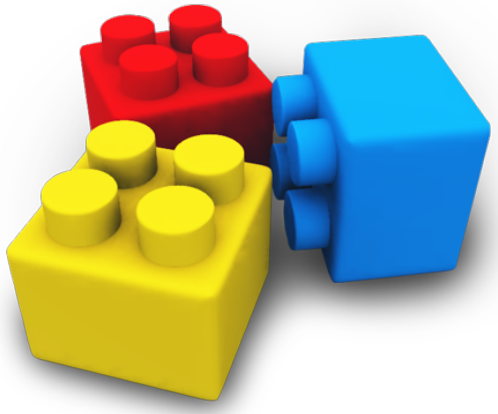
Built-in exec task, e.g:

```
task foo(type: Exec) {
  commandLine '/bin/ls'
}
```

PLUGINS

The design philosophy of Gradle is that all useful features are provided by plugins, which can:

- Add new tasks to the project.
- Provide a default configuration for the added tasks. The default configuration adds new conventions to the project (e.g. the location of source code files).
- Add new properties which are used to override the default configuration of the plugin.
- Add new dependencies to the project.

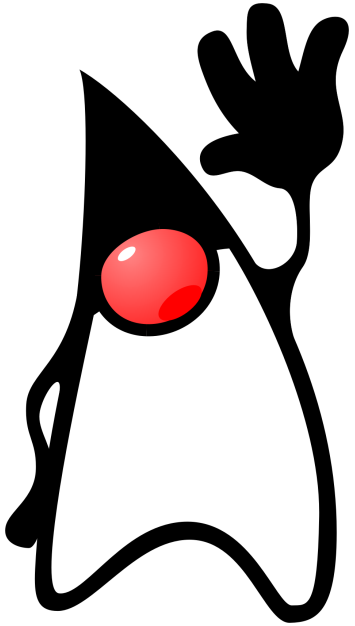


PLUGINS - STANDARD PLUGINS

Plugins	Description
java	Java compilation, testing and bundling capabilities.
war	Assembling web application WAR files.
groovy	Building Groovy projects.
scala	Building Scala projects.
maven	Publishing artifacts to Maven repositories.
checkstyle	Performs quality checks on your project’s Java source files
pmd	Performs quality checks on your project’s Java source files
sonar	Performs quality checks on your project’s Java source files

And many more plugins. You can even implement your own.

JAVA PLUGIN



We can create a Java project by applying the **java plugin** to our `build.gradle` file.

The Java plugin adds new conventions (e.g. the default project layout), new tasks, and new properties to our build. The default project layout is the following:

- The `src/main/java` directory contains the source code of our project.
- The `src/main/resources` directory contains the resources (such as properties files) of our project.
- The `src/test/java` directory contains the test classes.
- The `src/test/resources` directory contains the test resources.

All output files are created under the `build` directory, with the following subdirectories:

- The `classes` directory contains the compiled `.class` files.
- The `libs` directory contains the jar or war files created by the build.

JAVA PLUGIN - TASKS

The Java plugin adds many tasks to our build but the tasks which are relevant for this presentation are:

- `assemble` task compiles the source code of our application and packages it to a jar file. This task doesn't run the unit tests.
- `build` task performs a full build of the project.
- `clean` task deletes the build directory.
- `compileJava` task compiles the source code of our application.
- etc.

We can get the full list of runnable tasks and their description by running the following command at the command prompt:

```
$ gradle tasks
```

```
$ gradle clean build
:clean UP-TO-DATE
:compileJava
:processResources UP-TO-DATE
:classes
:jar
:assemble
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test UP-TO-DATE
:check UP-TO-DATE
:build

BUILD SUCCESSFUL
```

JAVA PLUGIN - EXAMPLE

Our build script must create an executable jar file from the following source (Homer.java):

```
package com.acme.simpsons;

public class Homer {
  public static void main(String[] args) {
      System.out.println("D'oh!");
  }
}
```

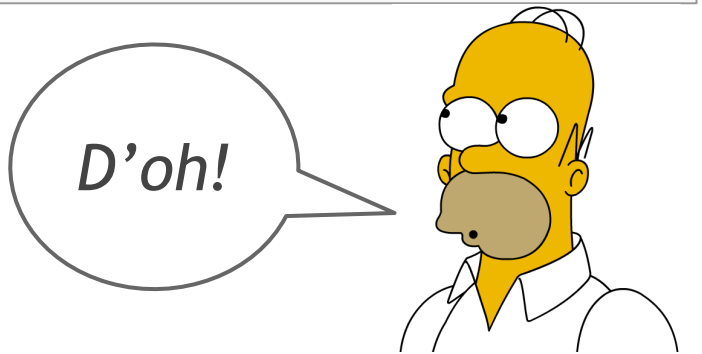
The generated jar should work like this:

```
$ java -jar build/libs/homer-1.0.jar
"D'oh!"
```

To achieve this, create the following `build.gradle` and execute the `gradle clean build` command:

```
apply plugin: 'java'

jar {
  baseName = 'homer'
  version = '1.0'
  manifest {
    attributes 'Main-Class': 'com.acme.simpsons.Homer'
  }
}
```



JAVA PLUGIN - DEPENDENCY

Add logging and unit testing capabilities:

```
apply plugin: 'java'

repositories {
  mavenCentral()
  maven { url "http://repo.maven.apache.org/maven2" }
  maven { url "http://repo.spring.io/libs-snapshot" }
  //ivy { url "http://repo.acme.com/repo" }
}

dependencies {
  compile 'org.slf4j:slf4j-api:1.7.5'
  // junit >= 4.0 is required
  testCompile group: 'junit', name: 'junit', version: '4.+'
}
```

Gradle originally used Ivy under the hood for its dependency management. Gradle has replaced this direct dependency on Ivy with a native Gradle dependency resolution engine which supports a range of approaches to dependency resolution including both POM and Ivy descriptor files.

Dependency configurations:

- **compile** - Required to compile the production source of the project.
- **runtime** - Required by the production classes at runtime. By default, also includes the compile time dependencies.
- **testCompile** - Required to compile the test sources. By default, also includes the compiled production classes and the compile time dependencies.
- **testRuntime** - Required to run the tests. By default, also includes the compile, runtime and test compile.

BEHIND THE SCENES

Those `task`, `apply`, `repositories`, and `dependencies` keywords are just normal methods. They are defined in the `Project` class. Once you know this, more things start to get clearer.

- `Task task(String name)` or `Task task(Map<String,?> args, String name)`
  - `type`: The class of the task to create.
  - `dependsOn`: A task name or set of task names which this task depends on
  - etc.
- `void apply(Map<String,?> options)` - Configures this project using plugins or scripts. Options:
  - `plugin`: The id or implementation class of the plugin to apply to the project.
  - etc.
- `void repositories(Closure configureClosure)` - Configures the repositories for this project.
- `void dependencies(Closure configureClosure)` - Configures the dependencies for this project.
  - This method executes the given closure against the DependencyHandler for this project. The DependencyHandler is passed to the closure as the closure's delegate.

More information: https://gradle.org/docs/current/javadoc/org/gradle/api/Project.html

SUMMARY

ADVANTAGES

- Compact DSL on top of Groovy
- Combines best parts from Ant and Maven
- Good support and big user base
- Many plugins

DISADVANTAGES

- On very big projects it can be slower than Ant or Maven
- Build scripts can become unreadable if conventions are not followed
- It can be difficult to understand the magic behind the scenes

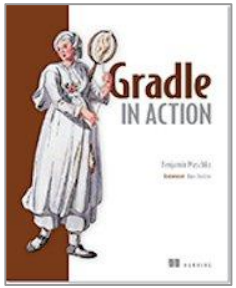
USEFUL RESOURCES

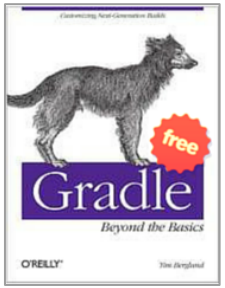
ONLINE DOCUMENTATION

- *Gradle Build Language Reference:* http://gradle.org/docs/current/dsl/
- *Task and task types:* http://gradle.org/docs/current/dsl/org.gradle.api.Task.html
- *Standard Gradle plugins:* http://gradle.org/docs/current/userguide/standard_plugins.html

BOOKS

- *Gradle in Action* - Benjamin Muschko (Manning)
- *Building and Testing with Gradle* - Tim Berglund, Matthew McCullough (O'Reilly) (free)
- *Gradle Beyond the Basics* - Tim Berglund (O'Reilly) (free)





THANK YOU. QUESTIONS?