

Software development made serious

A quick guide through the wonders of dependency management, build automation, teamwork with distributed version control systems, and continuous integration.

Danilo Pianini

`danilo.pianini@unibo.it`

ALMA MATER STUDIORUM—Università di Bologna a Cesena

Seminar in Cesena

March 18, 2016 - Cesena (Italy)



- 1 Teamwork with DVCS
 - git
 - git flow
 - GitHub and Bitbucket
- 2 Build automation
 - Dependency management
 - Gradle minimalist tutorial
- 3 Continuous integration
 - Why continuous integration
 - drone.io
 - Deployment
 - Code quality control



Harder than it seems

Software development is no trivial task...

...and it gets harder and harder with the complexity of the software and the number of people involved.

Some of the criticalities

- Importing and keeping libraries up to date (dependency management)
- Making sure that anyone can build your software
- Ensuring new features don't break existing ones (regression testing)
- Recreating documentation and reports that reflect the current status of the project
- Making your artifacts available to others (deployment)
- Working with others collaboratively
- Ensuring code conventions among members
- Ensuring code quality

Outline

1 Teamwork with DVCS

- git
- git flow
- GitHub and Bitbucket

2 Build automation

- Dependency management
- Gradle minimalist tutorial

3 Continuous integration

- Why continuous integration
- `drone.io`
- Deployment
- Code quality control



Bits of history

- In April 2005, BitKeeper, the SCM Linux was developed with, withdrawn the free (as in beer) use
- No other SCM met the requirements of Torvalds
 - Performance was the *real* issue with such a code base
- Torvalds decided to write his own
- The project was successful, and Torvalds appointed maintenance to Hamano

Why the name

I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'git'. ^a

— Linus Torvalds

^aFrom the project Wiki. "git" is slang for "pig headed, think they are always correct, argumentative"

The git README.md file

GIT - the stupid content tracker

"git" can mean anything, depending on your mood.

- random three-letter combination that is pronounceable, and not actually used by any common UNIX command. The fact that it is a mispronunciation of "get" may or may not be relevant.
- stupid. contemptible and despicable. simple. Take your pick from the dictionary of slang.
- "global information tracker": you're in a good mood, and it actually works for you. Angels sing, and a light suddenly fills the room.
- "goddamn idiotic truckload of sh*t": when it breaks



Features

- Extremely UNIX-oriented: tracks stuff like UNIX file permits.
- Distributed development (the whole development history is copied locally)
- Diff-based history tracking
- Implicit file naming (history preserved on renames)
- Very strong support to non-linear development
- Written in C
- Approximately 10 times faster than Mercurial, 100 times faster than other DVCS (e.g. Bazaar)
- Uses existing protocols (ssh, http, ftp, rsync...)
- Pluggable merge strategies (defaults to recursive three-ways merge or octopus for 3+ heads)



Usage recap

If any of these terms is not clear, please ask, because they are taken for granted

- Commit
- Pull
- Push
- Merge
- Branch
- Fast-Forward
- Branch checkout
- Tag
- Remote



Outline

1 Teamwork with DVCS

- git
- **git flow**
- GitHub and Bitbucket

2 Build automation

- Dependency management
- Gradle minimalist tutorial

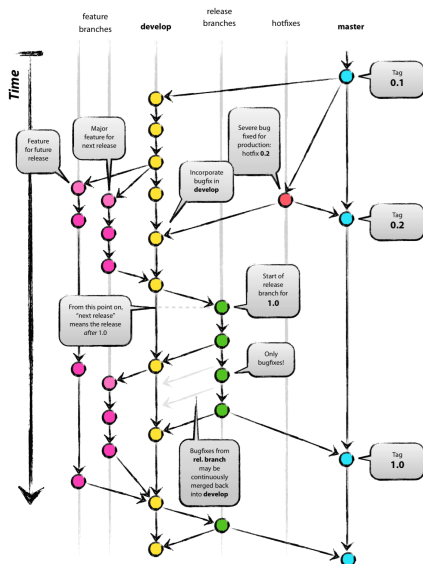
3 Continuous integration

- Why continuous integration
- `drone.io`
- Deployment
- Code quality control



Git Flow as a development model

- Mainline branch: `master`
 - Always alive
- Development branch: `develop`
 - Branches from `master`
 - Never merges
- New feature: `feature-*`
 - Branches from and merges to `develop`
- New release: `release-*`
 - Branches from `develop`
 - Merges to `master` and `develop`
 - Creates a new version
- Fix a bug on mainline: `hotfix-*`
 - Branches from `master`
 - Merges to `master` and `develop`
 - Creates a new version



Developing new features

When a new feature must be added to the software, the team responsible for it should branch a `feature-myfeature` branch from `develop`.

- `git checkout -b feature-myfeature develop`

Once all the work has been done, the branch should get merged in `develop` and deleted. Even if fast-forward, the merge should be visible, to prevent history loss.

- `git checkout develop`
- `git merge --no-ff feature-myfeature`
- `git branch -d feature-myfeature`

In order to minimize the merging effort, it is a good practice to incorporate changes from `develop` from time to time (e.g. when another team completed another feature).



Releasing a new version

When the status of the develop branch is (besides the very last bits) the status of the next production release, a `release-version` should branch from `develop`.

- `git checkout -b release-version develop`

In this branch, only things like version number changes or last minute bug fixes should get incorporated. Once done, we merge the `release-version` branch back into `develop`...

- `git checkout develop`

- `git merge --no-ff release-version`

...and `master`. Plus, we tag `master`, so that we keep a reference to the exact repository status at release time. Then, we delete the `release-version` branch.

- `git checkout master`

- `git merge --no-ff release-version`

- `git tag -a version`

- `git branch -d release-version`



Fix severe bugs affecting the mainline

You spot a bug in your current production branch. The fix must be delivered immediately.

Start a new `hotfix-version` branch:

- `git checkout -b hotfix-version master`

Change the version number, fix the bug (also add a regression test). Once done, repeat the procedure already seen for a normal release.

- `git checkout develop`

- `git merge --no-ff hotfix-version`

- `git checkout master`

- `git merge --no-ff hotfix-version`

- `git tag -a version`

- `git branch -d hotfix-version`



git flow

- This workflow, first suggested by Vincent Driessen, got very popular.
- The command sequence is repetitive, and got automated
- A git extension (not part of the git distribution) is available:
 - Introduces the `git flow` subcommand
 - Starts and finishes feature, release, hotfix (and support) branches
 - Under the hood, it calls exactly the commands listed previously
- My suggestion
 - *learn* git flow as a development model
 - Get acquainted with it using standard `git`
 - When you are very confident that you know what the tool is doing with your repository, use `git flow`
 - This is a good approach in general to new tools:
 - understand the idea
 - learn the basics
 - understand what goes on under the hood
 - use the advanced features productively



Back to centralized?

- There is no inherent concept of “central repository” in git
- The Git flow branching model considers a “central” repository (truth repo) where every developer insists
- The methodology focusses on how to maintain the central repo, however:
 - Each developer may have his own copy (fork) of the project
 - There may be a responsible of the central repository, the only one with write permits
 - Other developers request the maintainer to pull from their forks
 - there is specific support for this workflow in hosting platforms
 - The maintainer decides when to actually create releases
- In git, use `remote` to easily work with multiple forks



Outline

1 Teamwork with DVCS

- git
- git flow
- **GitHub and Bitbucket**

2 Build automation

- Dependency management
- Gradle minimalist tutorial

3 Continuous integration

- Why continuous integration
- `drone.io`
- Deployment
- Code quality control



GitHub

GitHub and Bitbucket are web based DVCS repositories, providing also a set of useful complementary features

Common features

- Support for Git repositories hosting
- Free for public, open source projects
- One issue tracker and one wiki per-project
- Pull request support
- Support for defining teams / organizations
- Code highlighting
- History navigation
- Markdown readme files



GitHub

Why GitHub

- By far the largest host of source code in the world
- One static website per project, one per user, one per organization
 - The `gh-pages` branch of each repository is implicitly considered a documentation web page
 - Support for Jekyll! ^a
- Markdown supported also in comments and issue posts
- Beautiful development statistics
- De facto standard for open source software

^a<https://jekyllrb.com/>



Bitbucket

Why Bitbucket

- Support for Mercurial repositories
- Unlimited private repositories for academic accounts (including `studio.unibo.it`)
- Arbitrary number of forks per project (GitHub is limited to one)
- Excellent for developing private projects, e.g. scientific papers



Outline

- 1 Teamwork with DVCS
 - git
 - git flow
 - GitHub and Bitbucket
- 2 Build automation
 - Dependency management
 - Gradle minimalist tutorial
- 3 Continuous integration
 - Why continuous integration
 - drone.io
 - Deployment
 - Code quality control



The concept of dependency

- Any software **depends** on other software
 - All the runtime base libraries (think of `java.lang.*` and `System.*`)
 - All the other libraries
 - Possibly, external resources (e.g., images, sounds, translation files...)
- Normally, this software depends on other software
 - That depend on other software
 - That depend on other software, and so on...
- A normal applications has a **tree** of dependencies



Example: print titles

Requirements

Write a program that:

- Visits TheTVDB.org (public TV Series database)
- Searches for a series
- Download the titles of all the episodes
- Prints them on screen

Questions

- Estimate how much code you'd need to write
- How much code can be just inherited from existing, public libraries?



Maybe less code than you may expect

code

```
package it.unibo.ci;

import ...

public final class PrintSeries {
    private static final String LANG = "it";
    private static final String SERIE = "Breaking Bad";
    private PrintSeries() {}

    public static void main(final String... args) throws IOException {
        final String key = IOUtils.toString(PrintSeries.class
            .getResourceAsStream("/TheTVDBAPIKey"), Charsets.UTF_8);
        final TheTVDBApi api = new TheTVDBApi(key);
        api.searchSeries("Breaking Bad", LANG).stream()
            .filter(s -> s.getSeriesName().equals(SERIE))
            .map(Series::getId)
            .flatMap(sId -> api.getAllEpisodes(sId, LANG).stream())
            .map(Episode::getEpisodeName)
            .forEach(System.out::println);
    }
}
```

Trick revealed

I used a few existing libraries!

- Google Guava
 - Used for referencing the UTF-8 Charset without using Strings (less error-prone)
- Apache Commons I/O
 - Used for converting a resource stream pointing to a String
- Omertron's thetvdbapi
 - Queries TheTVDB given a valid API key

But wait, there is more!

I only needed three libraries to get the job done. But are those libraries using other libraries?



The actual dependency tree

```
+--- com.google.guava:guava:+ -> 19.0-rc2
+--- commons-io:commons-io:+ -> 2.4
\--- com.omertron:thetvdbapi:+ -> 1.7
    +--- org.slf4j:slf4j-api:1.7.9
    \--- org.yamj:api-common:1.2
        +--- org.apache.commons:commons-lang3:3.3.2
        +--- commons-dbcp:commons-dbcp:1.4
            | \--- commons-pool:commons-pool:1.5.4 -> 1.6
        +--- commons-pool:commons-pool:1.6
        +--- commons-codec:commons-codec:1.10
        +--- org.apache.httpcomponents:httpclient:4.3.6
            | +--- org.apache.httpcomponents:httpcore:4.3.3
            | +--- commons-logging:commons-logging:1.1.3
            | \--- commons-codec:commons-codec:1.6 -> 1.10
        \--- org.slf4j:slf4j-api:1.7.9
```

- Libraries depend on other libraries
- All the libraries must be in the classpath!



Towards a dependency hell

This was a toy program, consisting of a single Java source file of some twenty lines of code.

Regardless, it requires **12** external libraries in order to run.

Libraries explosion

- It is very common for a non-toy project to get past 50 dependencies
 - Alchemist, not the biggest thing ever, counts 83 dependencies
- Hard to search, download and verify compatibilities by hand
- Version conflicts soon arise
 - one of your direct dependencies uses library A at version 1
 - another uses library A at version 2
 - you have a so-called transitive dependency conflict on A
- Upgrading by hand is even harder

Trust me, you want an automated tool to get you out of this hell.

Moar automation

Dependency management is just the *first* need that arises.

What you actually want to automatize

- Dependency management
 - Search and download from a reliable source your dependencies given a range of compatible versions
 - Search and download compatible transitive dependencies
 - Resolve version conflicts
- Automatically build, possibly on a clean instance of an OS, your software
- Run your test suite on your freshly built software
- Generate all the documentation that can be automatically generated, including build reports
- Package your software (and possibly also source and documentation)
- Stage, sign, release your software

Apache Maven

Idea

Provide a default build behaviour, declaratively describe how the specific build differs.

Facts

- The project is described by a Project Object Model (POM) file, in XML format
- The build *lifecycle* is composed of predefined *phases*
- Widely used
- Includes a dependency resolver
- Created a de-facto standard for the deployment of Java (or other JVM languages) artifacts

Apache Maven

Pros

- Transitive dependency management
- Build configuration inheritance
- Very easy to set up if the project follows the Maven way

Cons

- XML is verbose and not particularly human-friendly
- Little to no support for languages other than Java
- Pluggin-in non-standard behaviour is difficult, and often boils down to writing your own plugin



Outline

- 1 Teamwork with DVCS
 - git
 - git flow
 - GitHub and Bitbucket
- 2 Build automation
 - Dependency management
 - **Gradle minimalist tutorial**
- 3 Continuous integration
 - Why continuous integration
 - drone.io
 - Deployment
 - Code quality control



Gradle

Idea

Pick the best of Maven

- Dependency resolution
- Rich default configuration

Pick the best from “imperative” build systems, such as Apache Ant.

- Extreme flexibility

Facts

- The build is written in a Groovy based DSL
- Inherits from Ant the concept of “task”, replacing “lifecycle phases”
- Automagic resolution of the order in which tasks should be executed
- Incremental builds
- Supports many languages (Java, Scala, Groovy are first class citizens)

Gradle

Details

- Created in 2008 by Gradleware
- Mostly implemented in Java 5, with an outer layer in Groovy
- Free - both as in freedom (Apache License 2.0) and as in beer (no fees required)
- Source code available on GitHub
- Thorough documentation - though some advanced use requires some good personal initiative...



Gradle concepts

Project - from the Gradle documentation

What a project represents depends on what it is that you are doing with Gradle. For example, a project might represent a library JAR or a web application. It might represent a distribution ZIP assembled from the JARs produced by other projects.

A project does not necessarily represent a thing to be built. It might represent a thing to be done, such as deploying your application to staging or production environments.

Task - from the Gradle documentation

Each project is made up of one or more tasks.

A task represents some atomic piece of work which a build performs. This might be compiling some classes, creating a JAR, generating Javadoc, or publishing some archives to a repository.

Under the hood

- The Gradle build script is *technically* a valid Groovy script (if you consider the Gradle API)
- Anything that has not a valid Groovy syntax is not a valid Gradle build script
- Groovy makes it easy to create DSLs, Gradle is built relying on such a feature
- Everything you write is actually proper Groovy code (method calls, closures, and so on), but (you'll see) that aspect is very nicely hidden
- At the high level, the feeling is to just have to configure an existing plugin, much like Maven, for most of the things you normally do
- When needed, it is easy to get at the lower level and fiddle with the internals, possibly in an imperative fashion



My fist Gradle experiment

```
build.gradle
```

```
println 'Hello, World!'
```

Terminal

```
$ gradle
```

```
Hello, World!
```

```
:help
```

```
Welcome to Gradle 2.11.
```

```
To run a build, run gradle <task> ...
```

```
To see a list of available tasks, run gradle tasks
```

```
To see a list of command-line options, run gradle --help
```

```
To see more detail about a task, run gradle help --task <task>
```

```
BUILD SUCCESSFUL
```

```
Total time: 1.598 secs
```

Lessons learned

- Gradle executes `build.gradle` as a Groovy script
- Any valid Groovy script is also a valid `build.gradle`



Print the project name

```
build.gradle
```

```
println name
```

Terminal

```
$ gradle
```

```
01 - Project Property
```

```
:help
```

```
Welcome to Gradle 2.11.
```

```
To run a build, run gradle <task> ...
```

```
To see a list of available tasks, run gradle tasks
```

```
To see a list of command-line options, run gradle --help
```

```
To see more detail about a task, run gradle help --task <task>
```

```
BUILD SUCCESSFUL
```

```
Total time: 2.342 secs
```

Lessons learned

- A valid `build.gradle` is not in general a valid Groovy script (it has access to the whole Gradle API)
- `name` is a property of the `Project` object, and as such is valid and available
- If nothing better is available, the project name is taken from the directory name where `build.gradle` is located
- For a complete overview of the `Project` object properties, refer to the Gradle documentation



Set the project name

build.gradle

```
version = '0.0-test'  
println name  
println version  
println rootProject.version
```

settings.gradle

```
rootProject.name = 'continuous-integration-seminar'
```

Terminal

```
$ gradle  
continuous-integration-seminar  
0.0-test  
0.0-test  
:help  
  
[...blah blah blah...]
```

Lessons learned

- In `build.gradle`, modifiable properties for the `Project` object can be set and accessed
- `settings.gradle` can be used to override the default project settings for read-only global properties, such as the project name
- `rootProject` is a reference to the main project – there may be a hierarchical project structure, and `rootProject` refers to the outer (in our case is the only one)



Project configuration

build.gradle

```
println "Starting the build for ($group:$name) at version $version"
println "$projectDescription"
println "$projectLongName is licensed under $licenseName"
```

settings.gradle

```
rootProject.name = "$artifactId"
```

gradle.properties

```
group = it.unibo.pslab
artifactId = testproject
version = 0.0.0
projectLongName = Seminar on Continuous integration
projectDescription = An introductory seminar on the wonders of continuous integration
licenseName = Creative Commons 3.0 NC-BY-SA
```

Project configuration

Terminal

```
$ gradle
```

```
Starting the build for (it.unibo.pslab:testproject) at version 0.0.0
```

```
An introductory seminar on the wonders of continuous integration
```

```
Seminar on Continuous integration is licensed under Creative Commons 3.0 NC-BY-SA
```

```
:help
```

```
Welcome to Gradle 2.11.
```

```
To run a build, run gradle <task> ...
```

```
To see a list of available tasks, run gradle tasks
```

```
To see a list of command-line options, run gradle --help
```

```
To see more detail about a task, run gradle help --task <task>
```

```
BUILD SUCCESSFUL
```

```
Total time: 2.622 secs
```

Lessons learned

- `gradle.properties` is a standard Java property file
- `gradle.properties` can store any variable, in case a variable name matches a project property, such property will get overridden (if writable)
- `settings.gradle` also has access to `gradle.properties` definitions
- Any property can be accessed from within a bracket-enclosed string by prefixing it with the `$` symbol
- Gradle offers high flexibility in configuring a build – with great flexibility comes great design responsibility



My first Gradle task

build.gradle

```
task myTask {
    doLast {
        println "$projectLongName is licensed under $licenseName"
    }
    doFirst {
        println "Starting the build for ($rootProject.group:$rootProject.name) at
                                                    version $version"

        println "$projectDescription"
    }
}
```

Terminal

```
$ gradle myTask
:myTask
Starting the build for (it.unibo.pslab:testproject) at version 0.0.0
An introductory seminar on the wonders of continuous integration
Seminar on Continuous integration is licensed under Creative Commons 3.0 NC-BY-SA

BUILD SUCCESSFUL
```

Lessons learned

- Tasks are written in the `build.gradle` script
- Tasks can be invoked as `gradle` subcommands
- Every task has a list of actions, and when it executes it runs all of them in the order they are declared
- The default task has only actions `doFirst` and `doLast`
- Under the hood, we implemented the `Task` interface (refer to the Gradle documentation), overriding the default behaviour of actions `doLast` and `doFirst` with our own closures



Dependencies among tasks

build.gradle

```
task first << {
    println "Starting the build for ($rootProject.group:$rootProject.name) at ver
                                                sion $version"
}

task thenAfter << {
    println "$projectDescription"
}

task last {
    dependsOn thenAfter
    doFirst {
        println "$projectLongName is licensed under $licenseName"
    }
}

thenAfter.dependsOn(first)
```

Dependencies among tasks

Terminal

```
$ gradle first
:first
Starting the build for (it.unibo.pslab:testproject) at version 0.0.0

BUILD SUCCESSFUL

Total time: 3.095 secs
```

Terminal

```
$ gradle last
:first
Starting the build for (it.unibo.pslab:testproject) at version 0.0.0
:thenAfter
An introductory seminar on the wonders of continuous integration
:last
Seminar on Continuous integration is licensed under Creative Commons 3.0 NC-BY-SA

BUILD SUCCESSFUL

Total time: 1.958 secs
```

Lessons learned

- Tasks can declare dependencies among each other: one task may need other tasks to complete successfully
- The less dependencies are declared, the faster is the build (due to parallelization)
- Groovy flexibility allows for operator overloading: `<<` is no longer left shift in Gradle, but “task injection”
- Look at the mixture of declarative and imperative parts in `task`
 - a declarative property comes first, that very much resemble a piece of configuration;
 - an imperative to-do comes after
- The properties for every task are evaluated *before* the task itself is executed (it would be too late to evaluate them when the task is already executing)



Minimal Java build

src/main/java/HelloWorld.java

```
public class HelloWorld {  
  
    public static void main(String[] args) {  
        System.out.println("Hello, world!");  
    }  
  
}
```

build.gradle

```
apply plugin: 'java'
```



Minimal Java build

Terminal

```
$ gradle build
: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

Total time: 3.882 secs

This build could be faster, please consider using the Gradle Daemon:
https://docs.gradle.org/2.11/userguide/gradle_daemon.html

Minimal Java build

Terminal

```
$ tree -A build
build
|-- classes
|   |-- main
|       |-- HelloWorld.class
|-- dependency-cache
|-- libs
|   |-- 00 - Basic.jar
|-- tmp
|   |-- compileJava
|   |-- jar
|       |-- MANIFEST.MF

7 directories, 3 files
```

Pretty magic, huh?



Lessons learned

Gradle Plugins

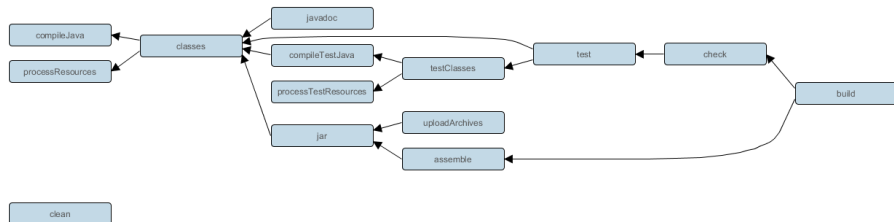
- Gradle at its core *intentionally* provides very little for real world automation. All of the useful features, are added by plugins.
- Plugins provide new concepts, configuration blocks, and tasks

Why plugins

- They can be reused across projects, lowering the maintaining burden
- Allow for higher modularization
- Allow for better encapsulation: good plugins host the whole imperative logic, so that in the user's `build.gradle` is left only only declarative configuration.
- Plugins are very easy to write (just implement `Plugin` in Groovy) and to share

Lessons learned

The Java plugin tasks



The Java plugin new concepts

Introduces the concept of *source set*, with an associated compile classpath and runtime classpath, with defaults that copy the Maven conventions:

- main containing `src/main/java` and `src/main/resources`
- test containing `src/test/java` and `src/test/resources`

Let's build our toy example

src/main/java/it/unibo/ci/PrintBreakingBad.java

```
package it.unibo.ci;

import java.io.IOException;
import org.apache.commons.io.IOUtils;
import com.google.common.base.Charsets;
import com.omertron.thetvdbapi.TheTVDBApi;
import com.omertron.thetvdbapi.model.Episode;
import com.omertron.thetvdbapi.model.Series;

public final class PrintBreakingBad {
    private static final String LANG = "it";
    private static final String SERIE = "Breaking Bad";
    private PrintBreakingBad() {}

    public static void main(String... args) throws ClassNotFoundException, IOException {
        final String key = IOUtils.toString(PrintBreakingBad.class.getResourceAsStream("/TheTVDBAPIKey"),
            Charsets.UTF_8);
        final TheTVDBApi api = new TheTVDBApi(key);
        api.searchSeries("Breaking Bad", LANG).stream()
            .filter(s -> s.getSeriesName().equals(SERIE))
            .map(Series::getId)
            .flatMap(s -> api.getAllEpisodes(s, LANG).stream())
            .map(Episode::getEpisodeName)
            .forEach(System.out::println);
    }
}
```

Let's build our toy example

build.gradle

```
apply plugin: 'java'

sourceCompatibility = "$jdkVersion"

repositories {
    mavenCentral()
}

dependencies {
    compile "com.google.guava:guava:$guavaVersion"
    compile "commons-io:commons-io:$commonsIoVersion"
    compile "com.omertron:thetvdbapi:$tvDbApiVersion"
}
```



Let's build our toy example

gradle.properties

```
group = it.unibo.apice
artifactId = printbbepisodes
version = 0.0.0
projectLongName = Breaking Bad Episode Titles
projectDescription = A program that fetches information about Breaking Bad on TheTVDb and prints episodes
licenseName = Apache License 2.0

jdkVersion = 1.8

commonsIoVersion = +
guavaVersion = 19.0
tvDbApiVersion = [1.6, 1.7]
```

settings.gradle

```
rootProject.name = "$artifactId"
```



Let's build our toy example

gradle.properties

```
$ gradle clean build
:clean
:compileJava
Download https://repo1.maven.org/maven2/com/google/guava/guava/19.0/guava-19.0.pom
Download https://repo1.maven.org/maven2/com/google/guava/guava-parent/19.0/guava-parent-19.0.pom
Download https://repo1.maven.org/maven2/com/omertron/thetvdbapi/1.7/thetvdbapi-1.7.pom
Download https://repo1.maven.org/maven2/org/sonatype/oss/oss-parent/9/oss-parent-9.pom
Download https://repo1.maven.org/maven2/org/slf4j/slf4j-api/1.7.9/slf4j-api-1.7.9.pom
...many transitive dependencies after...
Download https://repo1.maven.org/maven2/commons-dbcp/commons-dbcp/1.4/commons-dbcp-1.4.jar
Download https://repo1.maven.org/maven2/commons-pool/commons-pool/1.6/commons-pool-1.6.jar
Download https://repo1.maven.org/maven2/commons-codec/commons-codec/1.10/commons-codec-1.10.jar
Download https://repo1.maven.org/maven2/org/apache/httpcomponents/httpclient/4.3.6/httpclient-4.3.6.jar
Download https://repo1.maven.org/maven2/org/apache/httpcomponents/httpcore/4.3.3/httpcore-4.3.3.jar
:processResources
: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

Total time: 9.002 secs
```

Lessons learned

- Dependencies in Gradle can be specified by defining repositories and a list of packages this project depends on
- Maven Central and JCenter are first-class citizens (a configuration method is provided by the java plugin)
- Dependencies can be specified for the different project phase: `compile`, `test`, `runtime`, `testRuntime...`
 - Minimization of the exported dependency tree
- Dependencies can be specified with a specific version, a version range, or using `+` as “latest”
 - With `1.+` meaning “the latest version starting with 1.”
- Transitive dependencies are resolved automatically (if available in the repository, of course)
- Specifying the `sourceCompatibility` is not required, but recommended



Eclipse integration

A Gradle integration plugin is available for Eclipse (GREclipse).
Download it pointing to the correct update site and not through the market (the version there is outrageously out of date)

build.gradle

```
apply plugin: 'eclipse'

eclipse {
    classpath {
        downloadJavadoc = true
        downloadSources = true
    }
}
```

Downloads from the repositories also sources and javadocs and feeds them to Eclipse: much easier to develop...



More complex builds

- These basic elements, along with the Gradle documentation, should be sufficient for building more complicated products
- More advanced controls are available
 - Transitive dependency substitution and filtering
 - Personalised tasks based on the existing ones (e.g. create a runnable fat jar)
 - Usage of external program (e.g. rely on graphviz and doclet to enrich javadoc with generated UML schemas)
 - Create tasks that inject other tasks dynamically
- Use the Gradle documentation
- Ask for help (search first) on Stack Overflow
- Writing your own plugin is matter of collecting in a separate source file your own tasks and configuration methods
 - Extract it from the build, build it with Gradle, publish it
 - If published, you can use your own plugin to build your own plugin
 - Check out `org.danilopianini.SmarTrRR` for a very simple example



Outline

- 1 Teamwork with DVCS
 - git
 - git flow
 - GitHub and Bitbucket
- 2 Build automation
 - Dependency management
 - Gradle minimalist tutorial
- 3 Continuous integration
 - Why continuous integration
 - `drone.io`
 - Deployment
 - Code quality control



Automation failure

Build automatization is extremely important, but things can go wrong in very subtle ways.

“This has no impact anyway” failure

- Simplicio changes a single, seemingly irrelevant line of code.
- The build takes up minutes, so he just pushes the code.
- One month after, Salviati pulls from the mainline. The build fails.
- The failure is not trivial, so Salviati must bisect, wasting time.

⇒ Always run the full build, for **any** change.



Automation failure

“Works for me” failure

- Simplicio always runs the build on his own development system.
- The build succeeds, and Simplicio is done with his work.
- Salviati pulls from the mainline on a fresh PC. The build fails: a number of things may have gone wrong:
 - A local cache in Simplicio's system is keeping alive a package that does not exist anymore
 - A local cache is not triggering a dependency update that would break the build
 - There is a configuration file or a resource loading pointing to Simplicio's file system resources
 - A required resource is not in tracking
 - ...

⇒ Always run the build on a fresh, clean system (I am not kidding).

Continuous integration

Development practice:

- Commit *often*
- Keep in sync with the “truth repo” branch
- Run the full build in an automated fashion, on a fresh system **at every commit**
 - Keep the build fast!
- Make it easy to get the latest artifacts
- The build result must be available to all developers
- Automate deployment

A plethora of available products

- `drone.io`
- Travis CI
- Jenkins
- ...

Outline

- 1 Teamwork with DVCS
 - git
 - git flow
 - GitHub and Bitbucket
- 2 Build automation
 - Dependency management
 - Gradle minimalist tutorial
- 3 Continuous integration
 - Why continuous integration
 - **drone.io**
 - Deployment
 - Code quality control



drone.io

- Open source software, locally installable
- Online (closed source) service
- The free plan includes unlimited builds on unlimited public projects
- Integrated with GitHub and Bitbucket
 - Provides web hooks to trigger builds on push
- Supports Maven and Gradle
- Spawns each build on a fresh Ubuntu Linux machine
 - Modifiable environment variables
 - Instances of databases (MySQL / PostgreSQL / MongoDB...) available
 - The build is actually a sequence of bash commands
 - Java 8 version not so up to date :(
- If the build terminates successfully, the platform deploys (on a SSH server, Heroku, Amazon S3...)
- Email notifications for success / failures
- Selection of the artifact to publish



drone.io

Let's take a tour on my web browser...

The screenshot shows the drone.io web interface for a project named 'alchemist' at the repository 'git://github.com/DanySK/alchemist.git'. The interface includes a top navigation bar with links for 'Dashboard', 'Docs', 'Account', 'New Project', and 'Logout'. A left sidebar contains links for 'Build & Test', 'Deployments', 'Notifications', 'Artifacts', 'Status Badges', 'Repository', and 'Members'. The main content area features a 'Build Now' button with a dropdown arrow, accompanied by a light blue informational box that reads: 'Kickoff build request. Use this testing changes to your build script. Make sure to save changes first (Save is at the bottom)'. Below this, there are sections for 'Language' (set to 'Java 8'), 'Databases' (with checkboxes for 'MySQL' and 'PostgreSQL', and a 'more ...' link), and 'Environment Variables' (a large text input area). A note below the input area says 'Enter one per line (ex: FOO=bar)'. The 'Commands' section shows the 'working directory' as '/home/ubuntu/src/github.com/DanySK/alchemist' and a text area containing the commands:

```
./gradlew  
tar -cf build/reports/reports.tar build/reports/*.html
```

. A note below the commands says 'Enter one command per line (ex: echo foo)'. At the bottom of the main content area is a green 'Save' button.

Automation failure

“Build system not a dependency” failure

- Simplicio uses the super-fancy feature introduced in Gradle 2.42.
- Salviati and the rest of the team are still running on Gradle 2.11: the build fails for them.

⇒ The build system is itself a dependency of our software, and should be tracked as part of it



Gradle Wrapper

The Gradle wrapper

Gradle provides a builtin wrapper task:

- Downloads a gradle core jar and a few settings files
- Those file must be added to the tracker
- `gradle wrapper --gradle-version 2.11` downloads version 2.11
- `./gradlew <task>` runs `<task>` with that exact version
- It is a good practice to include the Gradle wrapper configuration in your `build.gradle` / `gradle.properties`

`build.gradle`

```
task wrapper(type: Wrapper) { gradleVersion = "$gradleVersionToUse" }
```

`gradle.properties`

```
gradleVersionToUse = 2.11
```

Automation failure

“But it was working” failure

- Simplicio has a working build, perfectly set up with the Gradle wrapper, and a correctly configured instance of drone.io
- After a couple of months, Salviati triggers the build, and it miserably fails, due to a dependency range now pointing to an incompatible artifact.

⇒ The build should be triggered *at least* once a day, to intercept problems early.



Night builds

- drone.io can be triggered via POST to start a build on the requested branch.
- This must be done externally (e.g. from a PC with a cronjob) — at least with the free account

trigger-build.sh

```
cat triggers | while read tr;
do for branch in "master" "develop"
do curl -X POST "https://drone.io/github.com/${tr}&branch=${branch}"
done
done
```

triggers

```
DanySK/javallib?token=UPERSECRET
DanySK/SmarTrRR?token=ANOTHERSECRET
DanySK/alchemist?token=TOKENNONONONONO
Protelis/Protelis?token=YETANOTHERAUTHCODE
```

Outline

- 1 Teamwork with DVCS
 - git
 - git flow
 - GitHub and Bitbucket
- 2 Build automation
 - Dependency management
 - Gradle minimalist tutorial
- 3 Continuous integration
 - Why continuous integration
 - `drone.io`
 - **Deployment**
 - Code quality control



Online repositories

The deployment of your software greatly varies depending on who's destined to.

Publicly available, open source software

- Very common by-product (and sometimes product) of academic research
- Just shipping source code on GitHub and / or publishing jar files on a random server **is not enough**
- Especially for libraries or middleware, it is **way** better to provide a proper repository, where other projects can just point to and import the product as Gradle / Maven / Ant / Ivy *dependency*
- Public services exist that offer a free and reliable repository



OSSRH — aka Maven Central

- Offered and managed by Sonatype
- Default software source for Maven builds
- Trivial to setup with any other build automation tool
- De-facto standard
- No-retract policy: if you publish an artifact, you cannot modify it, no exception allowed
- Gets copied from other repositories, e.g. jCenter
- Artifact staging and release through an instance of Sonatype Nexus
- Artifacts have a product name and belong to a group
- A Sonatype JIRA account and digital signature is required to manage a group
- Digital signature on artifacts required
- Strict quality control: sources and documentation must be provided
- See: <http://central.sonatype.org/pages/ossrh-guide.html>



Deployment automatization

We have our artifacts, automatically tested and compiled (at least) every day on our nice automatic build / test / integration framework.

We also want to have automatic deployment to OSSRH, so we need:

- ➊ Generation of all required artifacts: a sources jar file and a javadoc jar file
- ➋ An OSSRH account
- ➌ A GPG signature of all these artifacts
- ➍ Creation of a `pom.xml` file
- ➎ Automatic publication in our target repository
- ➏ **Manual** release of final versions of our software
 - You don't want a new official version per day with no change
 - You don't want a new, possibly inconsistent, version per commit
 - Deciding whether or not something should leave the stage or get promoted to release (still) requires human deliberation



Automatize artifact creation

Very easy in Gradle:

- A task that runs after classes are compiled (to be sure we don't pack non-compiling stuff) that fits in a jar all the source code
- A task that runs after the Javadoc has been generated, and compresses it in a jar file
- Configure Gradle to add those files to the generated artifacts

In build.gradle

```
task sourcesJar(type: Jar, dependsOn: classes) {
    classifier = 'sources'
    from sourceSets.main.allSource
}
task javadocJar(type: Jar, dependsOn: javadoc) {
    classifier = 'javadoc'
    from javadoc.destinationDir
}
artifacts {
    archives sourcesJar
    archives javadocJar
}
```

OSSRH account

- ❶ Create and publish your own GPG key
 - This is **personal**, and must not be shared
 - Good setup guide available at:
<http://central.sonatype.org/pages/working-with-pgp-signatures.html>
- ❷ Create a new account on Sonatype's Jira
- ❸ Create a new ticket, requesting a new group or to be added to an existing one
- ❹ In about two working days, you will get access to the repository, and your (signed) artifacts will be considered valid



Automatically sign artifact

This is the trickies part.

- 1 You can't sign on the `drone.io` server (unless you are fool enough to track your GPG keys).
- 2 You need to write your credentials to access Sonatype's Nexus, but can't do it on the `drone.io` server (unless, again, you are so totally crazy that you track a file with your account and password in)
- 3 You need to deploy your work on a trusted machine that hosts your GPG key and OSSRH user and password, sign from there, connect to Sonatype's Nexus from there, and upload the stuff.
- 4 You need a signing task, that must run only on the trusted machine, and get disabled elsewhere (or the build will fail)



efesto.apice.unibo.it

- 1 Arch Linux based virtual machine
- 2 Holds my GPG key and OSSRH credentials on an encrypted file system
- 3 Only exposes a passwordless SSH (only public key authentication)
- 4 Signs and uploads artifacts for Alchemist and Protelis
- 5 Hosts an HTTP service that exposes nightly builds
- 6 Also does the nightly build triggering
- 7 Little CPU and memory requirements
- 8 Not so reliable due to hardware management
- 9 If I had to do it again, I'd consider Amazon's / Google cloud...



Automatize artifact signing

There is a plugin for it

In build.gradle

```
apply plugin: 'signing'  
signing { sign configurations.archives }  
signArchives.onlyIf { Boolean.parseBoolean(signArchivesIsEnabled) }
```

In gradle.properties

```
signArchivesIsEnabled = false
```

In efesto.apice.unibo.it's ~/.gradle/gradle.properties

```
signing.keyId = my-key-id  
signing.password = my-super-secret-password-that-i-dont-share  
signing.secretKeyRingFile = /path/to/my/secring.gpg  
signArchivesIsEnabled = true
```

The local ~/.gradle/gradle.properties overrides properties set in the project's local gradle.properties



Automatize upload to OSSRH

There is a plugin for it!

In build.gradle

```
apply plugin: 'maven'

uploadArchives {
    repositories {
        mavenDeployer {
            beforeDeployment { MavenDeployment deployment -> signing.signPom(deployment) }
            repository(url: "https://oss.sonatype.org/service/local/staging/deploy/maven2/") {
                authentication(userName: ossrhUsername, password: ossrhPassword)
            }
            snapshotRepository(url: "https://oss.sonatype.org/content/repositories/snapshots/") {
                authentication(userName: ossrhUsername, password: ossrhPassword)
            }
        }
    }
    [NEXT SLIDE CONTENT HERE, WITH POM.XML GENERATION]
}
```



Automatize pom.xml generation

There is a plugin for it!

In build.gradle

```
[THIS GOES INSIDE mavenDeployer]
pom.project {
    name artifactId
    description projectDescription
    packaging 'jar'
    url "$scmRootUrl/$artifactId"
    licenses {
        license {
            name licenseName
            url licenseUrl
        }
    }
    developers {
        developer {
            name 'Danilo Pianini'
            email 'danilo.pianini@unibo.it'
            url 'http://danilopianini.apice.unibo.it/'
        }
    }
    scm {
        url "$scmRootUrl/$artifactId"
        connection "$scmType:$scmLogin/$scmRepoName"
        developerConnection "$scmType:$scmLogin/$scmRepoName"
    }
}
```

Variables for pom.xml generation and OSSRH upload

In gradle.properties

```
group = org.protelis
artifactId = protelis
version = 7.0.0
projectLongName = Protelis
projectDescription = Practical aggregate programming, hosted in Java
licenseName = GPL linking exception
licenseUrl = https://github.com/DanySK/protelis/blob/master/LICENSE.txt
scmType = scm:git
scmRootUrl = https://github.com/Protelis
scmLogin = git@github.com:DanySK
scmRepoName = alchemist-incarnation-protelis.git
ossrhUsername = do not write your real user here
ossrhPassword = do ABSOLUTELY NOT write your real password here
```

In efesto.apice.unibo.it's ~/.gradle/gradle.properties

```
ossrhUsername = your-ossrh-username
ossrhPassword = your-ossrh-password
```

Picking version numbers

Without compliance to some sort of formal specification, version numbers are essentially useless for dependency management. By giving a name and clear definition to the above ideas, it becomes easy to communicate your intentions to the users of your software.

— Semantic Versioning 2.0.0 (<http://semver.org/>)

Semantic versioning

- Formally described, with RFC-style wording
- Three levels plus optional: MAJOR.MINOR.PATCH[−OTHER]
 - MAJOR — Any incompatible API change
 - Major 0 is for initial development: anything may change at any time.
 - MINOR — Add functionality in a backwards-compatible manner
 - PATCH — Backwards-compatible bug fixes
 - OTHER — Optionally decorates the version with additional information.

Outline

- 1 Teamwork with DVCS
 - git
 - git flow
 - GitHub and Bitbucket
- 2 Build automation
 - Dependency management
 - Gradle minimalist tutorial
- 3 Continuous integration
 - Why continuous integration
 - drone.io
 - Deployment
 - Code quality control



Automatic code quality control

Why

- Immediately spot subtle bugs
- Early spot sub-optimal code (singular fields, missing finals...)
- Enforce encapsulation, spot cut/pastes (normally sign of bad design choices)
- Use a coherent style across your code
- Prevent style-change commits (“The Leopard commits”)
- Particularly important if there are many developers!



Automatic code quality control

How

- FindBugs
 - Analyzes the bytecode searching for bugs
- PMD
 - Source code analyzer for common programming flaws
 - Finds suspect cut/paste (CPD module)
 - Works for Java, Javascript, PLSQL, Velocity, XML, XSL
- Checkstyle
 - Forces code standard (indentation, formatting, Javadoc...)
 - Templates ready for Java
 - Can be configured for working with arbitrary files
 - Scalastyle for Scala is available

Plugins available for Eclipse and IntelliJ



Code quality control advices

- Ship the code checkers configuration files with your distribution
 - Usually just one or two XML files
- Make your IDE plugins configuration *match exactly* your build automation tool configuration
- Always track the IDE configuration with your SCM
 - New developers will import the configuration automatically if they have plugins installed
 - If a bad developer tries to change the configuration, you can spot it from the commit change set
- Pick the rule configuration that suits your project
 - Some rules are controversial at least
 - Some rule have default arbitrary limits
 - Alchemist rules come from several years of tuning, fill free to use them
- Require your developers to adhere



Run FindBugs with Gradle

In build.gradle

```
apply plugin: 'findbugs'
findbugs {
    ignoreFailures = true
    effort = "max"
    reportLevel = "low"
    excludeFilterConfig = resources.text.fromFile("findbugsExcludes.xml")
}
tasks.withType(FindBugs) {
    reports {
        xml.enabled = false
        html.enabled = true
    }
}
```



Run PMD with Gradle

In build.gradle

```
apply plugin: 'pmd'

pmd {
    ignoreFailures = true
    ruleSets = []
    ruleSetFiles = files("pmd.xml")
    targetJdk = pmdTargetJdk
    toolVersion = pmdVersion
}

tasks.withType(Pmd) {
    reports {
        xml.enabled = false
        html.enabled = true
    }
}
```

Run Checkstyle with Gradle

In build.gradle

```
apply plugin: 'checkstyle'
checkstyle {
    ignoreFailures = true
    configFile = new File("style.xml")
}
checkstyleMain << {
    ant.xslt(in: reports.xml.destination,
    style: new File("$project.projectDir/checkstyle-noframes-sorted.xsl"),
    out: new File(reports.xml.destination.parent, 'main.html'))
}
checkstyleTest << {
    ant.xslt(in: reports.xml.destination,
    style: new File("$project.projectDir/checkstyle-noframes-sorted.xsl"),
    out: new File(reports.xml.destination.parent, 'main.html'))
}
```



Summary of the whole process

