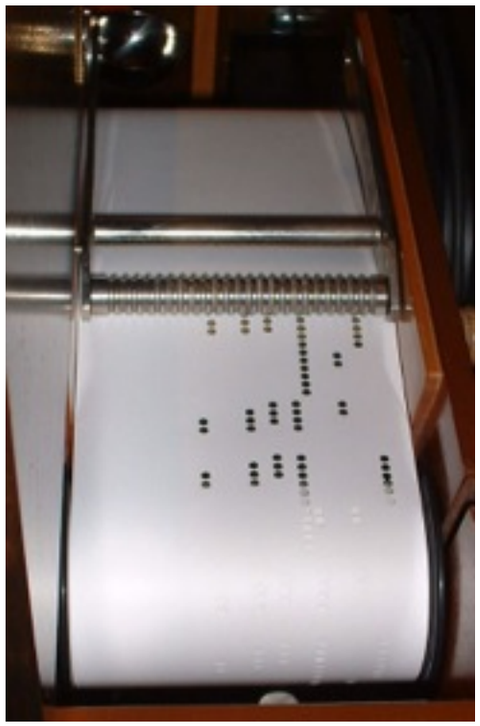
Why your build matters

Peter Ledbrook

e: peter@cacoethes.co.uk
w: http://www.cacoethes.co.uk
t: @pledbrook

When did it all begin?



- Programmable machines have a long history
- Mechanised instruments
- Looms

Punched cards

First used for looms in 18th Century France – Jacquard Loom 1801
IBM introduced punched card format 1928 – 80 columns!
    Text mode column width for DOS, Unix etc.
Imagine creating by hand
Keypunch machines (a little like typewriters) for creating them
    Errors mean throwing away the card and doing it again
Fortran and Cobol started with punched cards

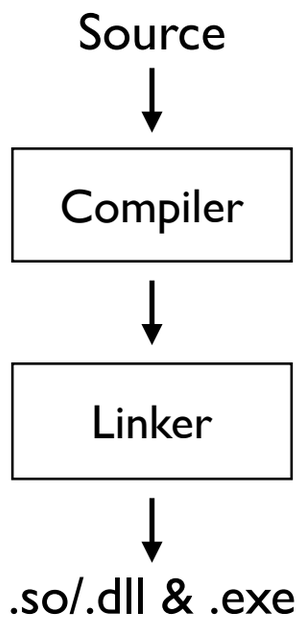
Fixing card decks

What if your cards are dropped on the floor?
This is an IBM 082 Sorter
Automation of a very error-prone manual process

Magnetism saves the day!

Finally, R/W data and code. The dawn of text editors.

C/C++



Multi-platform builds!

Make

```
*.c: *.o
    cc …

myapp: app.o mylib.o …
    ld …
```

+ manual custom dependencies

Focus mainly on compilation and linking
Make has powerful model based on file dependencies
Maintenance was a lot of work

Make

```
*.c: *.o
    cc …

myapp: app.o mylib.o …
    ld …
```

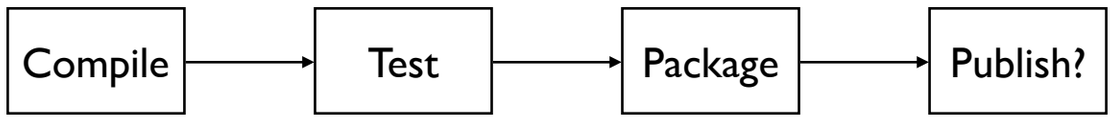
What's this?

+ manual custom dependencies

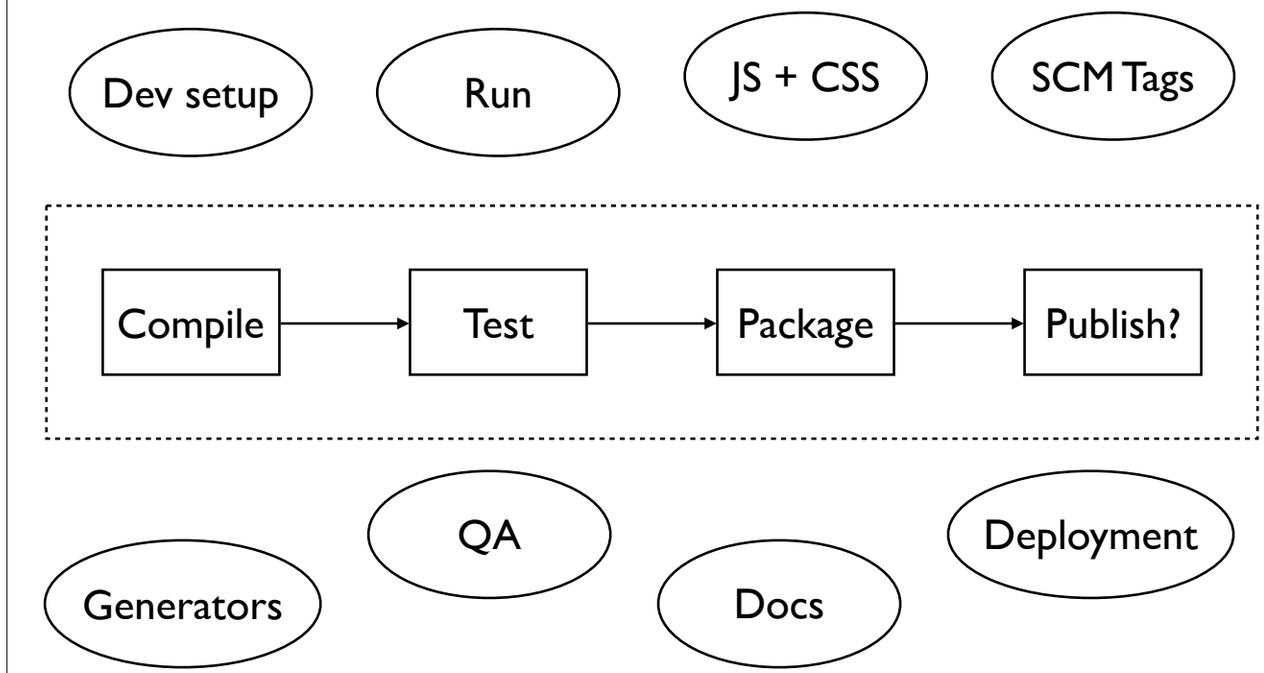
Java

- Compiler handles .java to .class dependencies
- JAR as ‘module’ unit
- Several standard deployment types
  - Applet
  - WAR
  - Standalone app

A standard build



Java build evolved

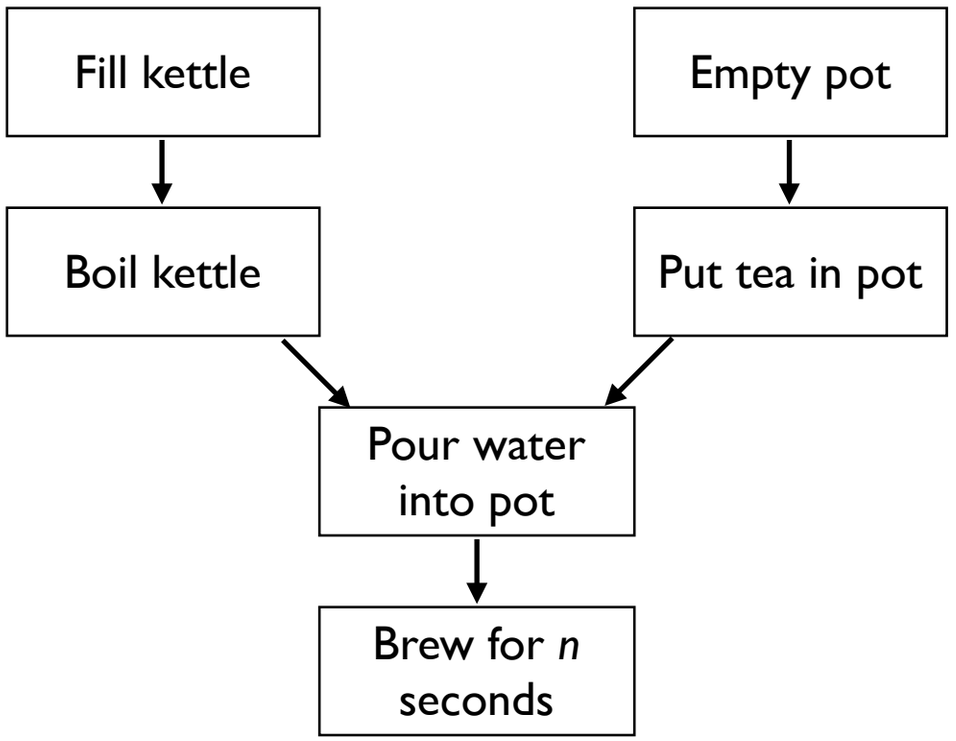


Builds are incorporating more and more steps

What will builds look like in 5 years time?

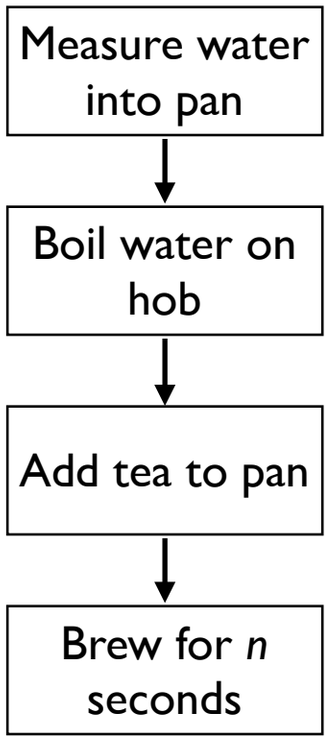
What is common across all these systems?
To understand that, let's look at a non-software process

Making tea



Clear that it's a series of steps (or tasks)
Some tasks require others to complete first
Others can run in parallel
Acyclic graph of tasks!

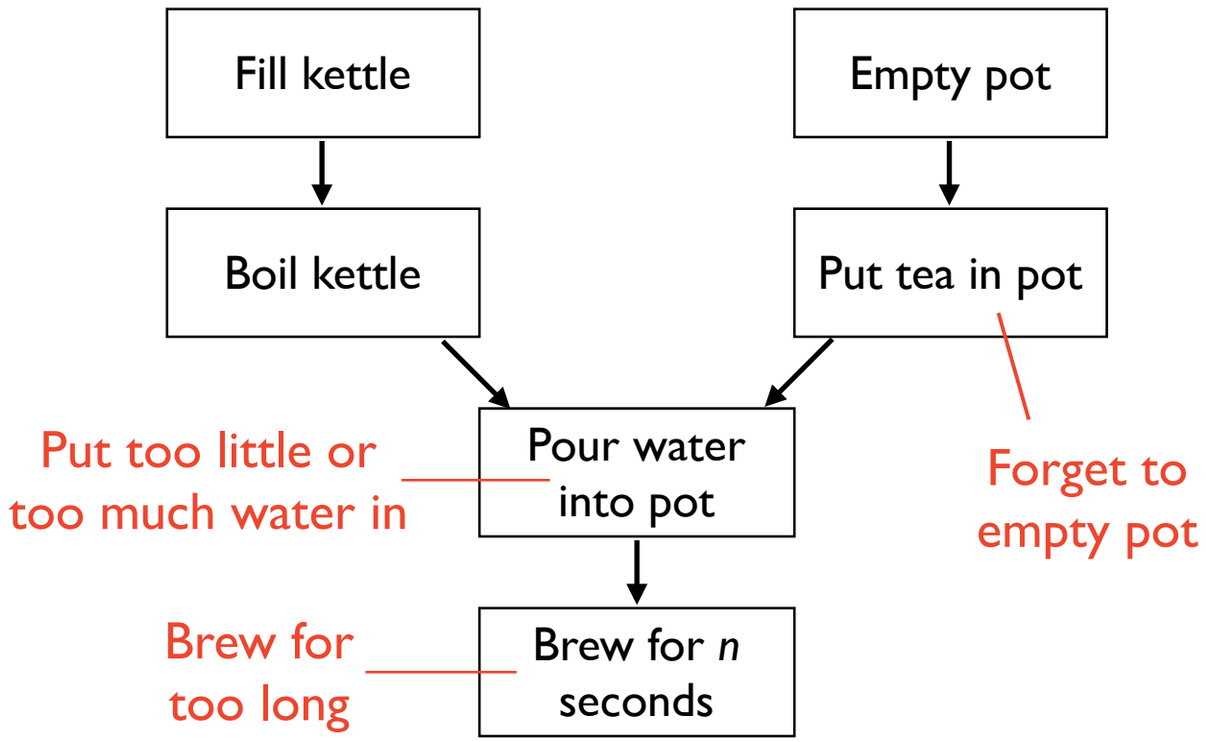
Making tea (my way)



A different way to make tea
Less standard, more appropriate for me

A standard process does not
mean an exclusive one

Many things can go wrong

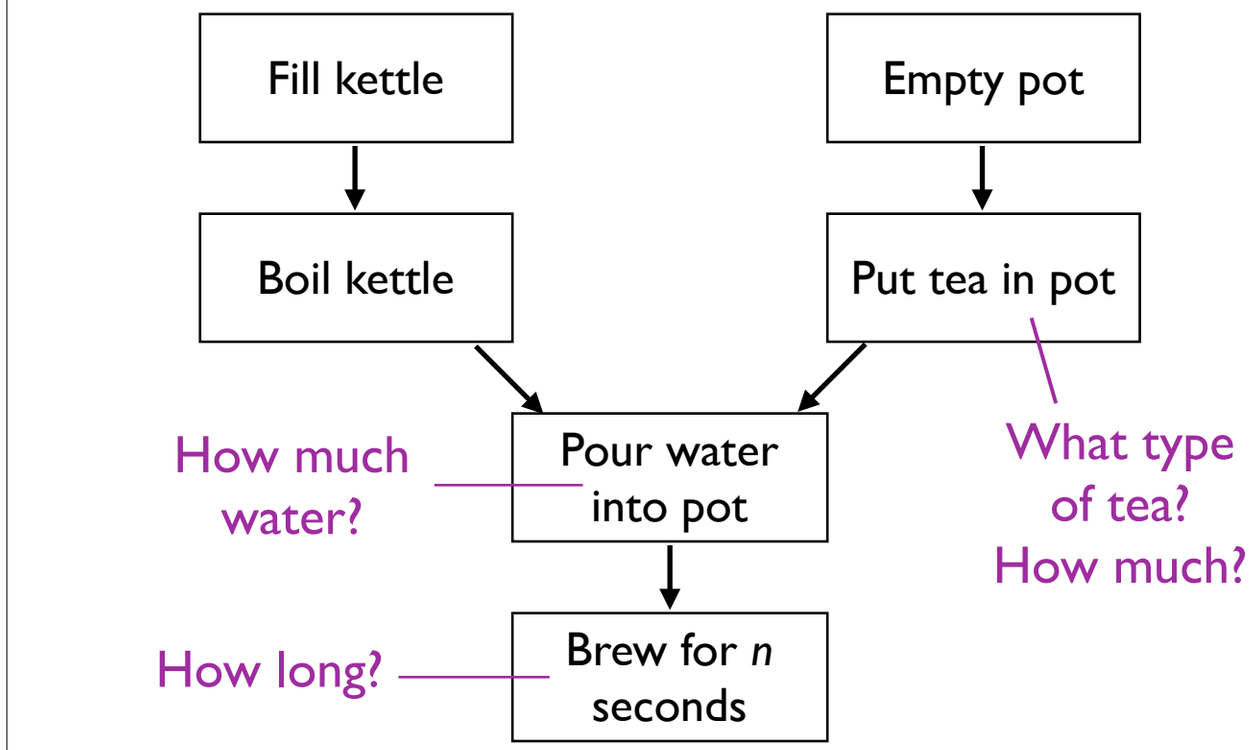


Automate to eliminate human error

People are fallible
Muscle memory can help
    e.g. tying shoe laces
    Not usually applicable in software development

What if your job depends on perfect tea every time?

Inputs & parameters



Build inputs:
- * water quality
- * tea bags/loose leaf
- * tea type

Build parameters:
- * quantity of tea
- * quantity of water
- * brew time

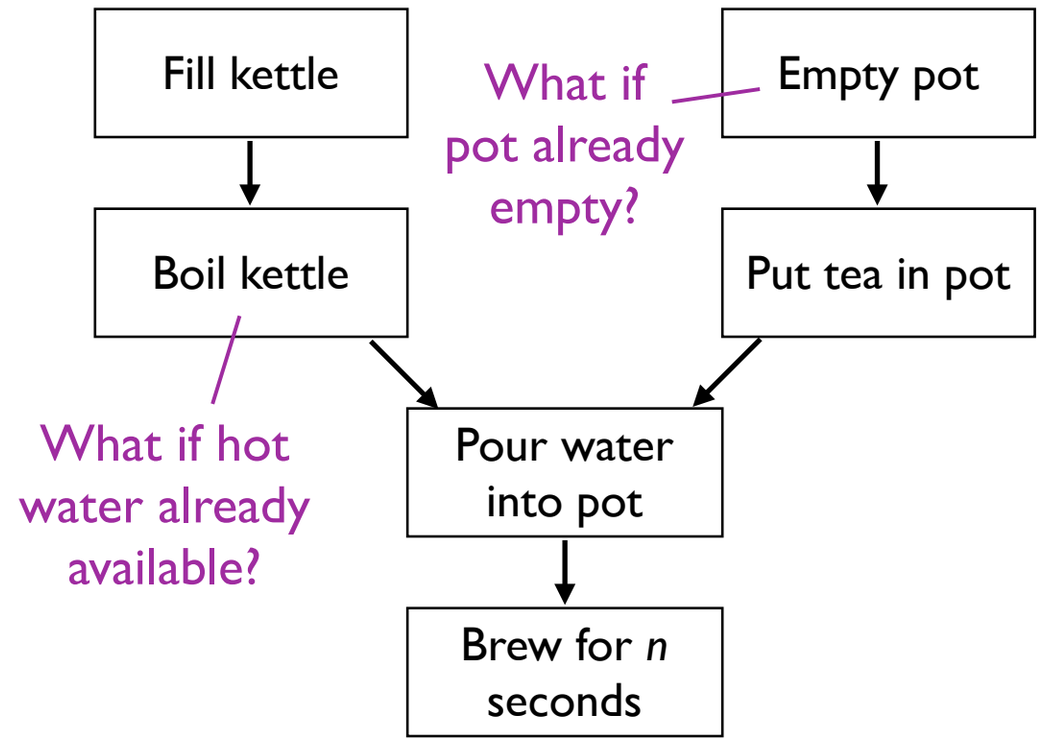
Same inputs should result in
same outputs

In other words, repeatable builds
Environment should not be a factor
- boiling kettle at top of mountain reduces temperature of water
Consider “works for me” with local Maven cache

How many cups of tea do you need to make a day?

Speed and efficiency are useful

Task avoidance



Doing all steps slows you down if some are unnecessary
Output of “boil kettle” is hot water
If hot water already exists, no need to boil

Incremental build (up-to-date checks) saves time

Often quicker to check whether a task is up to date rather than run it regardless

A build should be…

Automated

Repeatable

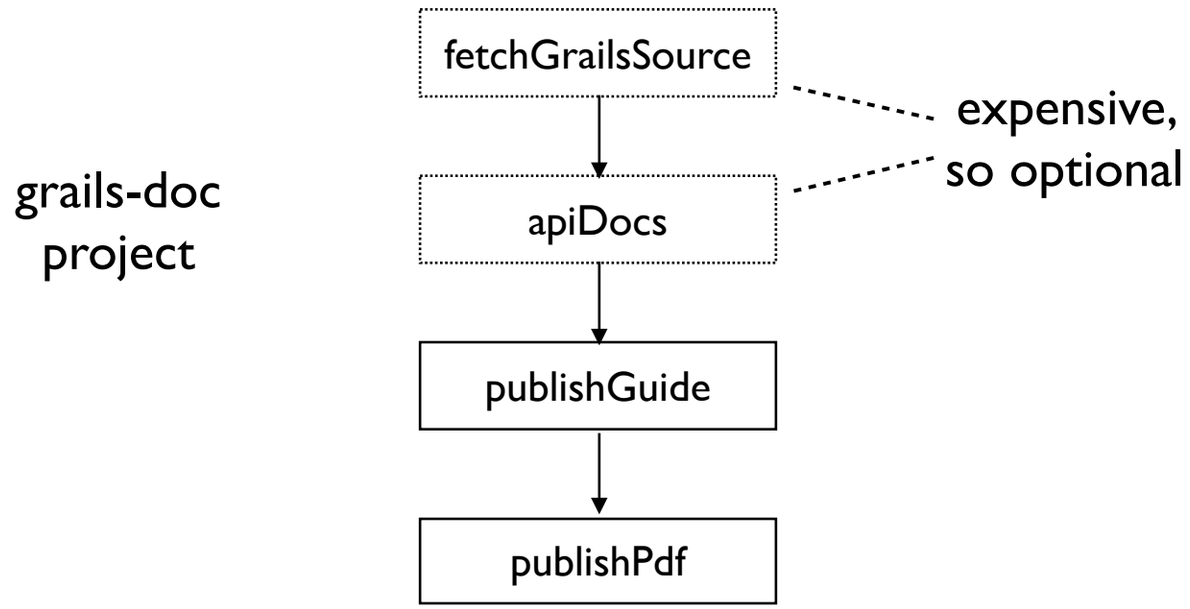
As fast as possible

The

Way

Automation

Custom build



A build to generate the Grails user guide
Automation of all steps
No standard tasks
Task graph is a suitable model

Custom tasks in Gradle

```
task publishGuide << {
    // Generate HTML from .gdoc
}
```

buildSrc/src/groovy/pkg/PublishGuideTask.groovy

`pkg.PublishGuideTask` in a JAR

Optional task

```
apiDocs.onlyIf {
   !System.getProperty("disable.groovydocs")
}
```

Add custom tasks to standard Groovy/Java projects

Even “standard” builds often have custom steps, e.g.

- integration tests
- deployment
- documentation

Stop using scripts or other tools separate from the build!

Repeatable builds

Depends on tasks

Are tasks environment-dependent?
Do system properties or environment variables have an effect?
What about number of cores?
Order of tests?
Gradle can’t help much with this - except for task ordering

Task ordering

- mustRunAfter/shouldRunAfter
- No task dependencies

```
task integTest {
    ...
}

task packageReports {
    mustRunAfter integTest
    ...
}
```

`packageReports` does not depend on `integTest`
But if both are executed, `packageReports` must come after `integTest`
shouldRunAfter is less strict - mostly to optimise feedback

Task ordering

- finalizedBy
- Ensures execution of the finalizer task

```
task integTest {
    finalizedBy packageReports
    ...
}

task packageReports {
    ...
}
```

`packageReports` does not depend on `integTest`
If `integTest` runs, then `packageReports` will run too, always after `integTest`

Dependency issues

Maven cache pollution (“works for me” syndrome)
Different remote repository configurations (where is the dependency coming from?)
Version resolution, eviction, and failed eviction
    Multiple JARs on the classpath and order counts

Gradle cache

- Origin checks
- Artifact checksums
- Concurrency safe
- Avoid `mavenLocal()`!

Artifacts are stored with source repo URL (allows origin checks)
Artifact checksums protect against different binaries with same name & version
Doesn’t solve the version conflicts issue

Resolution strategy

Defaults to ‘newest’ strategy

Fail the build if any version conflicts

```
configurations.all {
  resolutionStrategy {
    failOnVersionConflict()

    force 'asm:asm-all:3.3.1',
          'commons-io:commons-io:1.4'
}
```

Override version to use for specific dependencies

Fine-grained control over dependency versions
Automatic conflict resolution error-prone

Resolution strategy

```
configurations.all {
  eachDependency { details ->
    if (details.requested.name == 'groovy-all') {
      details.useTarget(
          group: details.requested.group,
          name: 'groovy',
          version: details.requested.version)
    }
  }
}
```

Control modules with
different names, same classes

A painful problem to debug

Debugging dependencies

```
gradle dependencies
```

```
gradle dependencyInsight --dependency ...
```

```
configurations.all.files.each { File f ->
  println f.name
}
```

`dependencies` gives you all dependencies in your project
`dependencyInsight` shows why/how a given dependency is included in the project
`<conf>.files` lists all the files and directories that will be on configuration's classpath

Fast build execution

Depends on tasks

How long do individual tasks take?
No parallel execution of tasks currently
Can build decoupled projects in parallel

Incremental build

```
:lazybones-app:compileJava UP-TO-DATE
:lazybones-app:compileGroovy UP-TO-DATE
:lazybones-app:processResources UP-TO-DATE
:lazybones-app:classes UP-TO-DATE
:lazybones-app:jar UP-TO-DATE
:lazybones-app:startScripts UP-TO-DATE
:lazybones-app:installApp UP-TO-DATE

BUILD SUCCESSFUL

Total time: 2.178 secs
```

Don’t execute tasks you don’t have to

Task inputs & outputs

```
class BintrayGenericUpload extends DefaultTask {
    @InputFile File artifactFile

    @Input     String artifactUrlPath

    @Optional
    @Input     String repositoryUrl
    ...
}
```

Use of annotations makes task support incremental build
Inputs and outputs can be values, files, directories

Who needs to use the build?

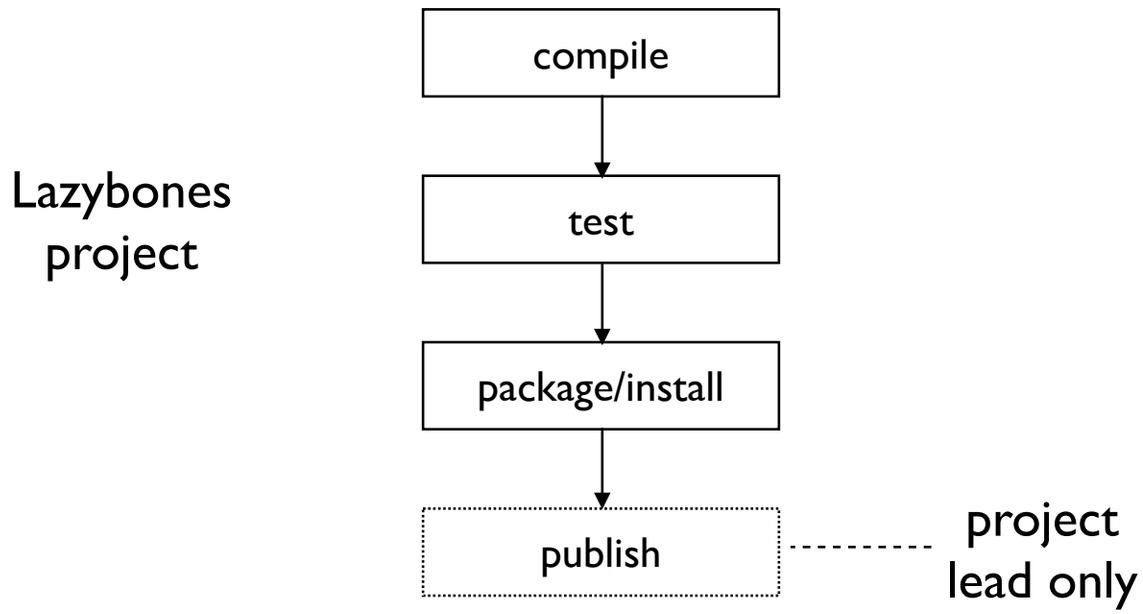
What do they need to use it?

Will they use the whole build?

How long does it take?

Developers, QA, production types
How much software needs to be set up before someone can use the build? How long is your wiki page?

User categories



If the project is published, require credentials
Fail fast
Other users don’t require credentials

Interrogate task graph

```
gradle.taskGraph.whenReady { graph ->
    if (graph.hasTask(":lazybones-app:uploadDist")) {
        verifyProperty(project, 'repo.url')
        verifyProperty(project, 'repo.username')
        verifyProperty(project, 'repo.apiKey')

        uploadDist.repositoryUrl = project.'repo.url'
        uploadDist.username = project.'repo.username'
        uploadDist.apiKey = project.'repo.apiKey'
    }
}
```

Only fail fast if `uploadTask` will be executed
Fail slow would require integration tests to run - adding minutes to deployment

Other uses

- Include class obfuscation conditionally
- Limit visibility of tasks based on role
- Update version based on ‘release’ task

Finally…

A build should have…

- Features
- Error reporting
- A model of the process

Just like any other piece of software!

Summary

- End-to-end process is the build
- 80-20 rule (80% standard 20% custom)
- Automate everything == save time
- Invest in build as if it's part of your code base
- Building software requires a rich model