Introduction to Gradle

The fundamentals of building projects with Gradle

Gradle

Install

You **must** have a JDK and the latest version of Gradle installed.

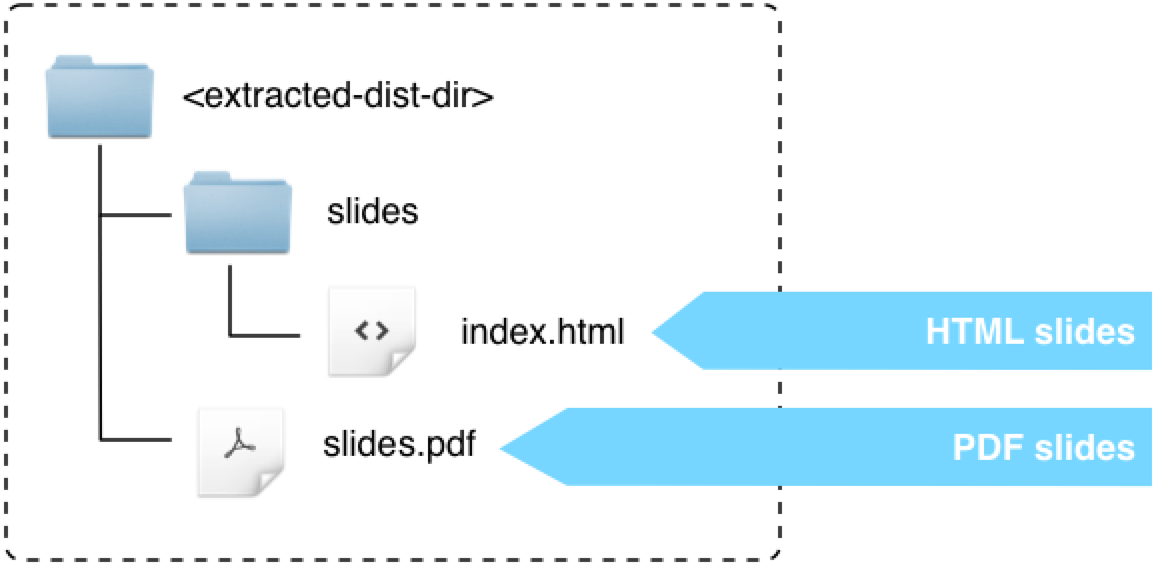
See the `setup-instructions.pdf` in your class materials for download links to install a JDK and Gradle.

HTTP Proxy

If you're behind a proxy, follow the setup instructions to configure Gradle to use your proxy.

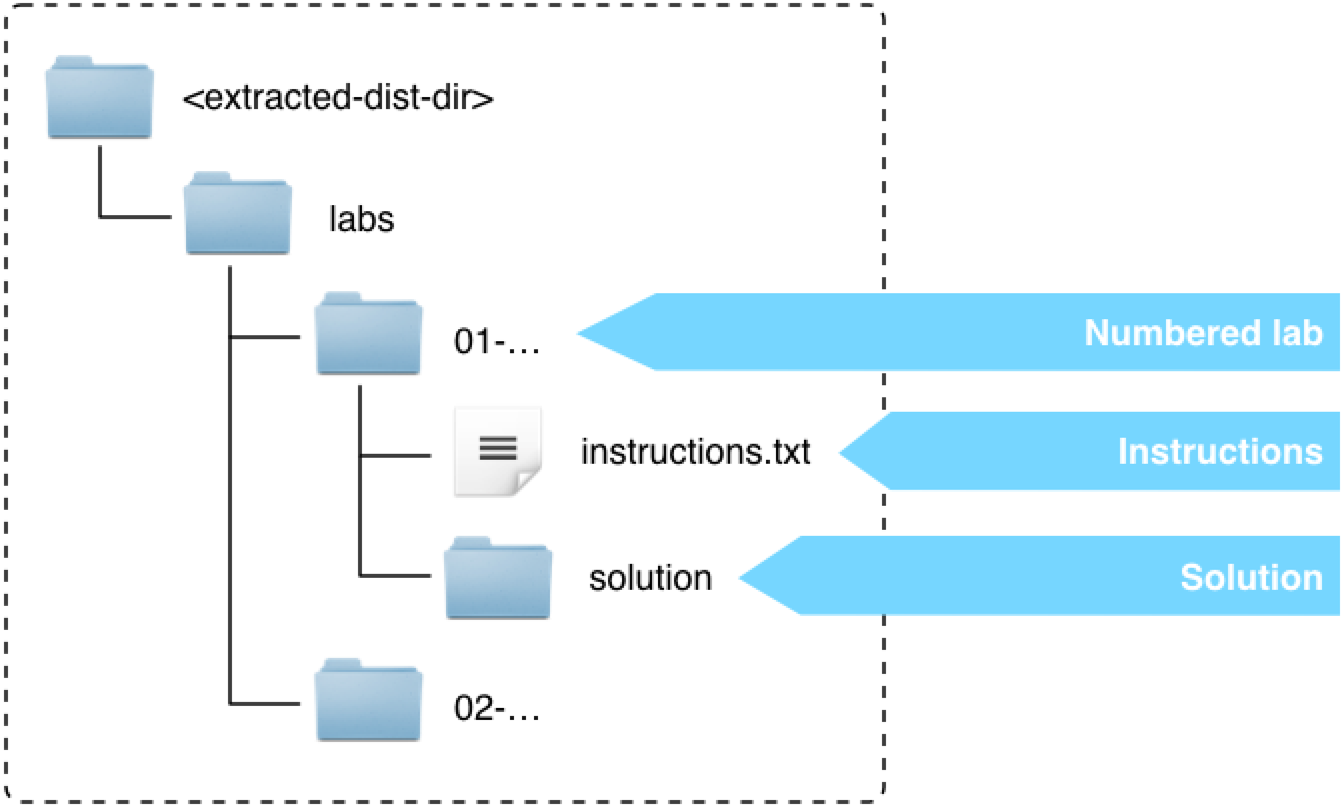
Slides

- Available in different formats
- Same content as today's presentation



Practical labs

- Solutions are available (but don't overuse them)
- Take your time and experiment
- The labs are not a test!



Objectives

- A solid understanding of basic Gradle concepts
- An ability to write and run simple Gradle tasks
- Knowledge of how to use the Java plugin
- Exposure to more advanced build capabilities

Ask questions

- Please ask questions at any time!

Topics we won't cover

- Android or other JVM language builds
- Continuous Integration/Delivery
- Plugin development
- Advanced dependency management techniques

Agenda

- About Gradle
- Gradle overview
  - Build scripts
  - Tasks
  - Working with files
  - Archives
- Building Java projects
- Dependency management basics
- Organizing a build
- More resources

Gradle

About the project

Gradle

Gradle is a build and automation tool.

Gradle can automate the building, testing, publishing and deployment of your software.

gradle.org

Gradle Project

- Open Source, Apache v2 license - Completely free to use
- Source code on Github - github.com/gradle/gradle
- Active user community, centered around discuss.gradle.org
- Frequent releases (minor releases roughly every 6-8 weeks)
- Strong quality commitment
  - Extensive automated testing (including documentation)
  - Backwards compatibility & feature lifecycle policy
- Developed by domain experts
  - Gradle, Inc. staff and community contributors

Gradle Documentation

- User Manual
    - Many chapters and self-contained downloadable samples
    - HTML
    - PDF
- Build Language Reference (gradle.org/docs/current/dsl/)
    - Best starting point when authoring build scripts
    - Javadoc-like, but higher-level
    - Links into Javadoc

Other Gradle Resources

- Install the latest release from gradle.org/install
- Older Gradle releases gradle.org/releases
- help.gradle.org
    - Portal to other resources

Running Gradle

Getting Information

Print command line options:

```
$ gradle -?
```

Print the available tasks in a project:

```
$ gradle tasks
```

Print basic help information:

```
$ gradle help
```

Getting help on a specific task

```
$ gradle help --task taskname
```

Best Practices for Running Gradle

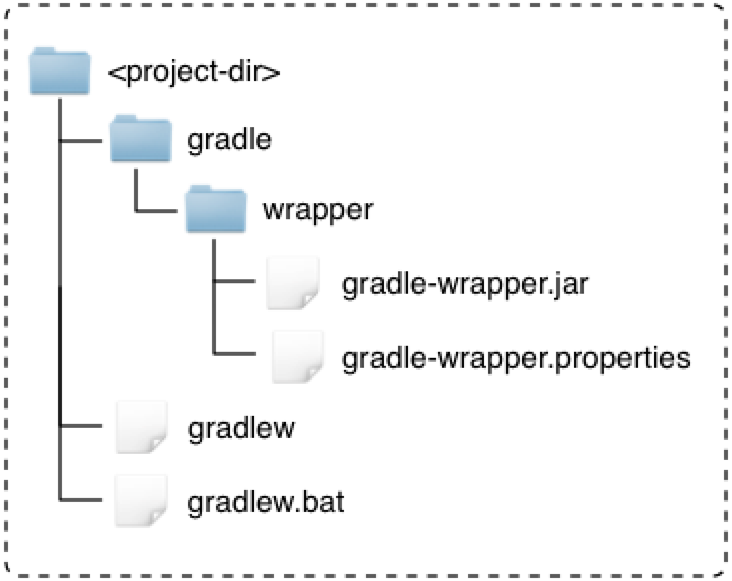
- Always use the wrapper
- Keep up-to-date with new releases
    - Performance bottlenecks are removed
    - New features are added
    - Deprecation warnings prevent surprises

Gradle Wrapper

Gradle Wrapper

A way to make sure everyone uses the same version of Gradle to build a project.

- **gradle-wrapper.jar:** Micro-library for downloading distribution
- **gradle-wrapper.properties:** Defines distribution download URL and options
- **gradlew:** Gradle executable for *nix systems
- **gradlew.bat:** Gradle executable for Windows systems
- Downloaded distributions go into your `GRADLE_USER_HOME/wrapper/dists` directory



Running a build with the wrapper:

```
$ ./gradlew build
```

The very first time you run a build with the wrapper, Gradle will download a copy of the distribution.

Wrapper task

- Bootstrap using the wrapper with your build with the `wrapper` task
- Wrapper task is built-in and generates:
    - wrapper scripts
    - wrapper jar
    - wrapper properties

```
$ gradle wrapper --gradle-version=6.6.1
```

- The `--gradle-version` flag lets you specify a particular version of Gradle to use.
- The `--distribution-type` flag lets you specify `all` if you want the complete distribution (the default is `bin`). The result is larger, but includes the source and documentation.

Lab

01-wrapper

Gradle Build Scans

Gradle

Creating build scans

- Creating a build scan is free.
- Build scans are a centralized and shareable record of a build.
- Build scans offer insight into how you are building your software.
- **All build scans created during this course will be uploaded to the public build scan service**.
- A self-hosted version is also available with more features.
- See the latest build scans for the Gradle project itself.

We encourage you to generate a build scan if you have a problem with a lab, so we can help you solve your problem. Just run your build with `--scan`.

Lab

02-create-build-scan

Gradle Basics

Gradle DSLs

Gradle is implemented in Java, with Kotlin and Groovy DSL layers.

Kotlin and Groovy bring:

- Domain Specific Language (DSL) capabilities
- Better readability and comprehensibility
- Many useful utilities built-in
- IDE support for build scripts (Kotlin in IntelliJ)

Gradle Build Scripts

- Files ending in `.gradle.kts` are compiled as Kotlin code
- Files ending in `.gradle` are compiled as Groovy code
- Build script files (`build.gradle[.kts]`) delegate to `org.gradle.api.Project`
- Settings script files (`settings.gradle[.kts]`) delegate to `org.gradle.api.initialization.Settings`
- The labs and slides only show Groovy DSL, but Kotlin DSL is very similar

Typical script files

Settings script

```
rootProject.name = 'name-of-build'

include "subproject"
include "another-subproject"
```

Build script

```
plugins {
    id 'java'
}

repositories {
    jcenter()
}

dependencies {
    implementation 'com.google.guava:guava:29.0-jre'
}
```

Configuration & Execution

Build Lifecycle

- Initialization Phase
  - Configure environment (init.gradle, gradle.properties)
  - Find projects and build scripts (settings.gradle)
- Configuration Phase
  - Evaluate all build scripts
  - Build object model (Gradle -> Project -> Task, etc.)
  - Build task execution graph
- Execution Phase
  - Execute (subset of) tasks

A *key* concept to grasp.

Tasks

Tasks are the basic unit of work in Gradle.

- created & configured by the user
- **executed by Gradle**

```
tasks.register("helloWorld") {
    doLast {
        println "Hello World!"
    }
}

// Old, but still valid syntax you'll see:
task helloWorld {
    doLast {
        println "Hello World!"
    }
}
```

All tasks implement the `Task` interface.

Task Actions

Tasks have a *list* of actions.

```
tasks.register("hello") {
  doLast {
    println "World!"
  }
  doFirst {
    println "Hello"
  }
}
```

Most tasks have one useful main action.

`doLast()` and `doFirst()` can be used to add actions to any task.

Discovering tasks

The built-in task `tasks` lists available tasks in a project, either defined in the build script or provided by applied plugins.

```
$ gradle tasks
```

Tasks in the output are organized by assigned `group` property e.g. `Build Setup` show up under the header `Build Setup tasks`.

The `description` property describes the purpose of a task.

Built-in tasks

Every Gradle project provides several tasks out-of-the-box.

Built-in tasks provide useful and commonly used functionality without having to apply any plugins.

Examples:

- `wrapper` - Generates the Wrapper files for this build.
- `help` - Demonstrates how to run Gradle from the command line.
- `dependencies` - Renders a tree of dependencies defined in the build.

Grouping tasks

By default, the tasks report only shows tasks that have been assigned a group.

```
tasks.register("hello") {
  group = 'Gradle Training"
  description = 'Prints a message.'

  doLast {
    println "Hello World!"
  }
}
```

```
> gradle tasks
...
Gradle Training tasks
---------------------
hello - Prints a message.
```

Hiding tasks

Task without a `group` property can be found by running `tasks --all`.

```
tasks.register("bye") {
    doLast {
        println "Bye World!"
    }
}
```

The `Other tasks` bucket lists all tasks without a group.

```
> gradle tasks --all
...
Other tasks
---------------------
bye
```

Lab

03-tasks

Fuzzy name matching

Save your fingers by only typing the bare minimum to identify a task.

```
tasks.register("myNameIsKindaLong") {
  doLast {
      println "long task!"
  }
}
```

```
> gradle mNIKL
```

Gradle understands how to match against camel-case task names. Ambiguous matches fail the build.

DSL Syntax and Tasks

```
// access existing task via its name
hello.dependsOn otherTask

// configure existing task via closure
hello {
  dependsOn otherTask
}

// configure new task
tasks.register("greet") {
  dependsOn otherTask
  doLast { println "Hello Gradler!" }
}
```

Tasks can be expensive to create, so Gradle has APIs to avoid creating and configuring tasks.

Task Types

You will usually use tasks of a certain type, that provide useful behavior (e.g. copy files).

```
tasks.register("copyFiles", Copy) {
    // Only configuration (actions are defined by the type)
    from('someDirectory')
    into('anotherDirectory')
}
```

Task is of type `Copy`. Configure it using its API.

If you don't specify a type, you get a `DefaultTask`.

Task Types and API

```
tasks.register("hello") {
    onlyIf { day == "monday" }
    doFirst { println "Hello" }
}
```

The `onlyIf()` method is a method of all tasks (i.e. part of `Task` interface).

```
tasks.register("copy", Copy) {
  from "someDir"
  into "anotherDir"
}
```

The `from()` method here is part of the `Copy` API.

A task's API allows you to *configure* the task.

Implementing Task Types

- POJO extending `DefaultTask`
- Declare action with `@org.gradle.api.tasks.TaskAction`

```
abstract class FtpTask extends DefaultTask {
  String host = "docs.mycompany.com"

  @TaskAction
  void ftp() {
    // do something complicated
  }
}
```

Lab

04-custom-tasks

Task Type > Ad-hoc Task

Prefer implementing task types to implementing ad-hoc tasks.

- Avoid global properties and methods
- Separate the imperative from the declarative
- Easy to refactor (e.g. from build script to Jar)
- Easier to utilize other Gradle features

Ad-hoc tasks are OK for small simple tasks.

Task Dependencies

- Tasks can depend on each other
- Semantic relationship (A produces something that B consumes)
- Executed tasks form a directed acyclic graph

```
tasks.register("foo")

// multiple ways to declare task dependencies
bar { dependsOn foo }
bar.dependsOn foo
```

Task Ordering

The order that tasks are executed in can be optimized.

```
tasks.register("unitTests") {}

tasks.register("integrationTests") {
    mustRunAfter unitTests
    // or: shouldRunAfter unitTests
}
```

`Task.mustRunAfter` - if this task executes, Gradle must run it after the given task. `Task.shouldRunAfter` - Weaker form of `mustRunAfter`. Gradle may run the tasks in another order if no other tasks are ready.

With no relationship between tasks, task order is undefined.

Task Finalization

Runs a task even if a preceding task has failed.

```
tasks.register("startWebServer") {}
tasks.register("stopWebServer") {}

tasks.register("integrationTests") {
    dependsOn startWebServer
    finalizedBy stopWebServer
}
```

Often used for releasing resources (cf. Java's try-finally).

Lab

05-task-dependencies

Working with the Filesystem

Files

- Primary function of most builds
- Standard Java File API
- Gradle adds new types (e.g `FileCollection`, `FileTree`)
- Fundamental to Gradle's input/output model

Gradle provides support for common operations out of the box (e.g. zip, copy, delete).

Project properties

Important file related properties:

- `projectDir` - the base directory of the project
- `buildDir` - the build output directory of the project
- `rootDir` - the base directory of the root project (multi-project)

The `buildDir` is `"$projectDir/build"` by default.

In plugins, don't assume this. Use `"$buildDir"`.

Relative files

Don't do this:

```
new File("src/main/java/Thing.java")
```

You don't know what the working directory of the JVM is.

Use:

```
project.file("src/main/java/Thing.java")
```

`Project.file(Object)` always resolves relative to the `projectDir`.

Many tasks accept `Object` for file types; resolved by `project.file()`.

Copy task

Copies files from one or more locations, to *one* destination.

```
tasks.register("copyLibs", Copy) {
  from "libsDir", "docs/index.html", "/some.txt"
  into "ide"
}
```

Powerful API, including filtering and transforming.

Multiple sources/sub directories

API has a tree like structure.

```
tasks.register("copyStuff", Copy) {
  exclude "**/.svn"  // default
  into "targetDir"
  // copies contents of sourceDir into targetDir/targetSubDir
  into("targetSubDir") {
    from "sourceDir"
  }
  into("targetSubDir2") {
    from "sourceDir2", "someFile.txt"
  }
  into("targetSubDir3") {
    from "sourceDir3"
    include "**/*.jpeg"
    exclude "**/obsoleteImages/*"
  }
}
```

Gradle

Transforming

Files can be mutated during copy.

```
tasks.register("copyStuff", Copy) {
  into "targetDir"
  from("someDir") {
    // Use Ant's HeadFilter
    filter(HeadFilter, lines: 25, skip: 2)
  }
  from("otherDir") {
    // Line by line transform
    filter { line -> line.substring(5) }
  }
  from("anotherDir") {
    // Groovy's SimpleTemplateEngine
    // "$foo" -> "bar", "$red" -> "blue"
    expand(foo: "bar", red: "blue")
  }
}
```

Renaming

Files can be renamed and/or moved.

```
tasks.register("copyStuff", Copy) {
  into "targetDir"
  from("someDir") {
    rename "(.*)_OEM_BLUE_(.*)", '$1$2'
  }
  from("otherDir") {
    eachFile { FileCopyDetails copyDetails ->
      if (copyDetails.name.length() > 10) {
        copyDetails.path = "longFileNames/$copyDetails.name"
      }
    }
  }
}
```

`eachFile` can also exclude files, deal with duplicates, etc.

Lab

06-copy

Permissions

Permissions at the destination can be specified.

```
tasks.register("copyStuff", Copy) {
  into("targetDir")
  into("bin") {
    from "src/bin"
    fileMode = 0755
    dirMode = 0755
  }
}
```

Particularly useful when creating archives (covered soon).

Sync Task

Same as `Copy`, except that destination will *only* contain copied files (and nothing else).

```
tasks.register("copyStuff", Sync) {
  from sharedNetworkLibsDir
  into "ide"
}
```

- Full copy (not incremental like rsync).
- [Sync.preserve](https://docs.gradle.org/current/dsl/org.gradle.api.tasks.Sync.html#org.gradle.api.tasks.Sync:preserve(org.gradle.api.Action)) can be used to keep files in the destination directory.
- Sync in the DSL reference

Archives

Archive Handling

Task type for each archive type (`Zip`, `Jar`, `War`, `Tar`).

- Similar to copy
  - Archiving: Copying to a directory
  - Unarchiving: Copying from a directory
- Supports transforming/renaming etc.

Archive Tasks

```
tasks.register("zipLibs", Zip) {
  into("ide") {
    from("libsDir", "docs/index.html")
  }
  from "src/license.txt"
}
```

Zip content:

- license.txt
- ide/someJarFromLibsDir.jar
- ide/index.html

Archive Names

Base plugin adds conventional naming defaults.

```
plugins {
    id "base"
}
tasks.register("zipLibs", Zip) {
  archiveBaseName = "services"
  // …
}
```

Pattern: *«archiveBaseName»-«archiveAppendix»-«archiveVersion»-«archiveClassifier».«archiveExtension»*

- archiveBaseName -> project.name
- archiveAppendix -> empty string
- archiveVersion -> project.version
- archiveClassifier -> empty string
- archiveExtension -> type extension

Default destinations

- Default destination dir for `Zip`/`Tar` (by `base` plugin)
    - `"build/distributions"`
- Default destination dir for `Jar`/`War` (by `java-base` plugin)
    - `"build/libs"`

Destination directory is customizable:

```
plugins {
    id "base"
}

tasks.register("myZip", Zip) {
  destinationDir = file("$buildDir/specialZips")
}
```

Unarchiving

Use `zipTree()` and `tarTree()` to specify archive *content*.

```
tasks.register("unpackArchives", Copy) {
  from zipTree("zip1.zip"), zipTree("jar1.jar")
  from(tarTree("tar1.tar")) {
    exclude "**/*.properties"
  }
  from "zip2.zip"
  into "unpackDir"
}
```

Merging

`zipTree()` and `tarTree()` can be used to merge archives.

```
tasks.register("mergedZip", Zip) {
  from zipTree("someZip.zip")
  from zipTree("otherZip.zip")
}
```

Shadow plugin is useful for building fat jars.

Plugins

Gradle Plugins

Plugins are just packaged build logic.

Plugins can do anything that you can do in a build script, and vice versa.

Plugins aid:

1. **Reuse** - avoid copy/paste
2. **Encapsulation** - hide implementation detail behind a DSL
3. **Modularity** - clean, maintainable code
4. **Composition** - plugins can complement each other

Typical Plugin Functions

Some of the things plugins typically do:

- **Extend the Gradle model** with new elements
- Configure the project according to **conventions**
  - Add new tasks
  - Configure existing model elements
  - Add configuration rules for future elements
- Apply some very **specific configuration**
  - Configure the project for very specific standards

Applying plugins

Plugins are applied in a `plugins` block:

```
plugins {
    id 'name-of-plugin'
}
```

Plugins can also have versions (if they are not built-in plugins)

```
plugins {
    id 'name-of-plugin' version '1.0'
}
```

Configuring where to find plugins.

Applying plugins (legacy)

Plugins can also be applied via `apply plugin:`

```
apply plugin: 'name-of-plugin'
```

This requires that the plugin already be added to the build script classpath.

```
buildscript {
    dependencies {
        classpath "plugin.group:name-of-plugin:1.0"
    }
}
```

The `plugins {}` block is preferable in most cases.

Building Java projects

Gradle

java-library Plugin

The basis of Java development with Gradle.

- "main" and "test" source set conventions
- Incremental compilation
- Dependency management
- JUnit & TestNG testing
- Javadoc generation

`api` vs `implementation`

Java libraries can [separate their implementation and API dependencies](#).

Dependencies appearing in the `api` will be transitively exposed to consumers of the library when compiling. Dependencies found in the `implementation` will not be exposed to consumers when compiling but will be available at runtime.

This has many advantages over a single `compile` time dependency scope.

Source Sets

A logical compilation/processing unit of sources.

- Java source files
- Non compiled source files (e.g. properties files)
- Classpath separation (compile & runtime)
- Output class files
- Compilation tasks

```
sourceSets {
  main {
    java {
      srcDir "src/main/java" // default
    }
    resources {
      srcDir "src/main/resources" // default
    }
  }
}
```

Lifecycle Tasks

The `java-library` plugin provides a set of “lifecycle” tasks for common tasks.

- `clean` - delete all build output
- `classes` - compile code, process resources
- `test` - run tests
- `assemble` - make all archives (e.g. zips, jars, wars etc.)
- `check` - run all quality checks (e.g. tests + static code analysis)
- `build` - combination of `assemble` & `check`

Testing

Built-in support for JUnit4, JUnit5 and TestNG.

- Pre-configured "test" task
- Automatic test detection
- Forked JVM execution
- Parallel execution
- Configurable console output
- Human-readable HTML reports
- Machine-readable reports for further processing (e.g. XML)

IDE integration

- Eclipse Buildship
- IntelliJ

IDEs can delegate to Gradle to run tests and other arbitrary tasks.

Lab

07-java-plugin

Dependency Management

Dependency Management

Gradle supports managed and unmanaged dependencies.

- “Managed” dependencies have identity and possibly metadata.
- “Unmanaged” dependencies are just anonymous files.

Managed dependencies are superior as their use can be automated and reported on.

Unmanaged Dependencies

```
dependencies {
  implementation fileTree(dir: "lib", include: "*.jar")
}
```

Can be useful during migration.

Managed Dependencies

```
dependencies {
  implementation "org.springframework:spring-core:5.2.8.RELEASE"
  implementation group: "org.springframework", name: "spring-web",
          version: "5.2.8.RELEASE"
}
```

Group/Module/Version

Configurations

Dependencies are assigned to *configurations*. See `java-library` defined configurations.

```
configurations {
  // default with "java-library" plugin
  compileOnly
  implementation
  runtimeOnly
  testCompileOnly
  testImplementation
  testRuntimeOnly
}

dependencies {
  implementation "org.springframework:spring-core:4.0.5.RELEASE"
}
```

See `Configuration` in DSL reference.

Transitive Dependencies

Gradle (by default) fetches dependencies of your dependencies. This can introduce version conflicts.

Only one version of a given dependency can be part of a configuration.

Options:

- Use default strategy (highest version number)
- Component metadata rules
- Dependency resolution rules
- Fail on version conflict
- Disable transitive dependency management
- Force a version
- Excludes

Fail on Conflict

Automatic conflict resolution can be disabled.

```
configurations {
  implementation {
    resolutionStrategy.failOnVersionConflict()
  }
}
```

If disabled, conflicts have to be resolved manually (using force, exclude etc.)

Cross Configuration Rules

Configuration-specific rules can be applied to all configurations.

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

`all` is a special keyword, meaning all things in the configuration container.

Disable Transitives

Per dependency…

```
dependencies {
  implementation("org.foo:bar:1.0") {
    transitive = false
  }
}
```

Configuration-wide…

```
configurations {
  implementation.transitive = false
}
```

Version Forcing

Per dependency…

```
dependencies {
  implementation("org.springframework:spring-core:4.0.5.RELEASE") {
    force = true
  }
}
```

Configuration-wide…

```
configurations {
  implementation {
    resolutionStrategy.force "org.springframework:spring-core:4.0.5.RELEASE"
  }
}
```

Excludes

Per dependency…

```
dependencies {
  testImplementation('org.spockframework:spock-core:1.0-groovy-2.4') {
    exclude module : 'groovy-all'
  }
}
```

Configuration-wide…

```
configurations {
  implementation {
    exclude module : 'groovy-all'
  }
}
```

Dependency Cache

Default location: `~/.gradle/caches/....`

- Multi-process safe
- Source location aware
- Optimized for reading (finding deps is fast)
- Checksum based storage
- Avoids unnecessary downloading
    - Finds local candidates
    - Uses checksums/etags

An opaque cache, not a repository.

Changing Dependencies

Changing dependencies are mutable.

Version numbers ending in `-SNAPSHOT` are changing by default.

```
dependencies {
  implementation "org.company:some-lib:1.0-SNAPSHOT"
  implementation("org:somename:1.0") {
    changing = true
  }
}
```

Default TTL is 24 hours.

Dynamic Dependencies

Dynamic dependencies do not refer to concrete versions.

```
dependencies {
  implementation "org.company:some-lib:2.+"
  // For ivy repositories, you can use Ivy symbolic versions.
  implementation "org:somename:latest.release"
}
```

Default TTL is 24 hours.

Controlling Updates & TTL

```
configurations.all {
  resolutionStrategy.cacheChangingModulesFor 4, "hours"
  resolutionStrategy.cacheDynamicVersionsFor 10, "minutes"
}
```

- `--offline` - don't look for updates, regardless of TTL
- `--refresh-dependencies` - look for updates, regardless of TTL

Dependency Reports

View the dependency graph.

```
$ gradle dependencies [--configuration «name»]
```

View a dependency in the graph.

```
$ gradle dependencyInsight --dependency «name» --configuration «name»
```

Built in tasks.

Repositories

- Any Maven/Ivy repository can be used
- Very flexible layouts are possible for Ivy repositories

```
repositories {
  jcenter()
  mavenCentral()

  maven {
    name "my co repo"
    url "https://repo.mycompany.com"
  }

  ivy {
    url "https://repo.mycompany.com"
    layout "gradle" // default
  }

  flatDir(dirs: ["dir1", "dir2"])
}
```

Lab

08-dependencies

Publishing

- Publish your artifacts to any Maven/Ivy repository
- Metadata file (`pom.xml`/`ivy.xml`) is generated
- Repository metadata (e.g. `maven-metadata.xml`) is generated
- Ivy Publish Plugin for publishing to Ivy repositories
- Maven Publish Plugin for publishing to Maven repositories

Publishing to Maven Repositories

```
plugins {
    id 'java-library'
    id 'maven-publish'
}

publishing {
  publications {
    maven(MavenPublication) {
      from components.java
    }
  }
  repositories {
    maven {
      url 'https://my.org/m2repo/'
    }
  }
}
```

- For Artifactory, JFrog provides an `artifactory-publish` plugin

Multi-project Builds

Multi-project Builds

- Flexible directory layout
- Project dependencies & partial builds
- Each project can use different plugins

Real world examples

- Spock
- Gradle

Defining a Multi-project Build

- Build structure is defined in `settings.gradle[.kts]`

```
// define the name of the build (defaults to directory name)
rootProject.name = "main"

// declare projects:
include "api", "shared", "services:webservice"

// by default, api subproject is in directory 'api'
project(":api").projectDir = file("/myLocation")

// by default: build files are "build.gradle" or "build.gradle.kts"
project(":shared").buildFileName = "shared.gradle"
```

Task/Project Paths

- All projects have a path that uniquely identifies them.
- Gradle uses `:` as a path separator.
- When running tasks on the command-line, you can combine the project path and the task name to select a task to execute.
  - `:` refers to the root project
  - `:clean` means to run the clean task in the root project only
  - `:api` refers to the api project
  - `:api:clean` means to run the clean task in the api project only

Implicit task selection

Running a task found only in subprojects from the root project will implicitly execute those tasks in the subproject

Runs clean in all subprojects:

```
$ gradle clean
```

Runs assemble in all subprojects:

```
$ gradle assemble
```

Runs test in all subprojects:

```
$ gradle test
```

Fuzzy name matching

Like tasks, fuzzy name matching works for project paths too.

If you had a project named `reallyLongName`, you could run clean in that project with:

```
$ gradle rLN:clean
```

Project Dependencies

Instead of using `group:name:version` to declare dependencies between projects, you can use project dependencies.

The method `project(String)` takes the path to the other project

```
dependencies {
  implementation "commons-lang:commons-lang:2.4"
  // Depends on the "shared" project
  implementation project(":shared")
}
```

Gradle automatically selects the publications from the other project.

Configuration Injection

Parent projects (including the root project) can injection configuration into subprojects.

```
// apply this configuration to all subprojects
subprojects {
  apply plugin: "java-library"
  dependencies {
    testImplementation "junit:junit:4.13"
  }
  test {
    jvmArgs "-Xmx512M"
  }
}
```

This should be used sparingly, since it can make it harder for you to understand all of the configuration that affects a project.

Composite builds

- Composite builds are a way to combine multiple builds into a single build.
- A composite build is made up of a root build and one or more "included builds"
- You can use composite builds to combine independently developed builds or decompose a large build into separate chunks.
- Learn more from the composite build samples.

Implicit `buildSrc` build

- `buildSrc` is a built-in included build.
- Just adding a `buildSrc` directory into the root of your project enables it.
- You can use this build to encapsulate and organize your build logic.
- `buildSrc` provides some built-in conveniences and automatically compiles and tests your build logic

Wrapping up

Performance features

Build caching

Gradle will skip execution of some work if it has been done before, even on other machines.

```
$ gradle build --build-cache
```

File system watching

Gradle will watch files on disk and skip to executing tasks more quickly.

```
$ gradle build --watch-fs
```

Parallel Builds

Run independent tasks from different projects in parallel.

```
$ gradle build --parallel
```

Useful features

Continue after Failure

```
$ gradle build --continue
```

Especially useful for CI builds.

Continuous Build

When the build completes, instead of exiting, watch the inputs of executed tasks and re-run the build when an input changes.

```
$ gradle build --continuous
```

Standard Gradle plugins

Gradle ships with many useful plugins.

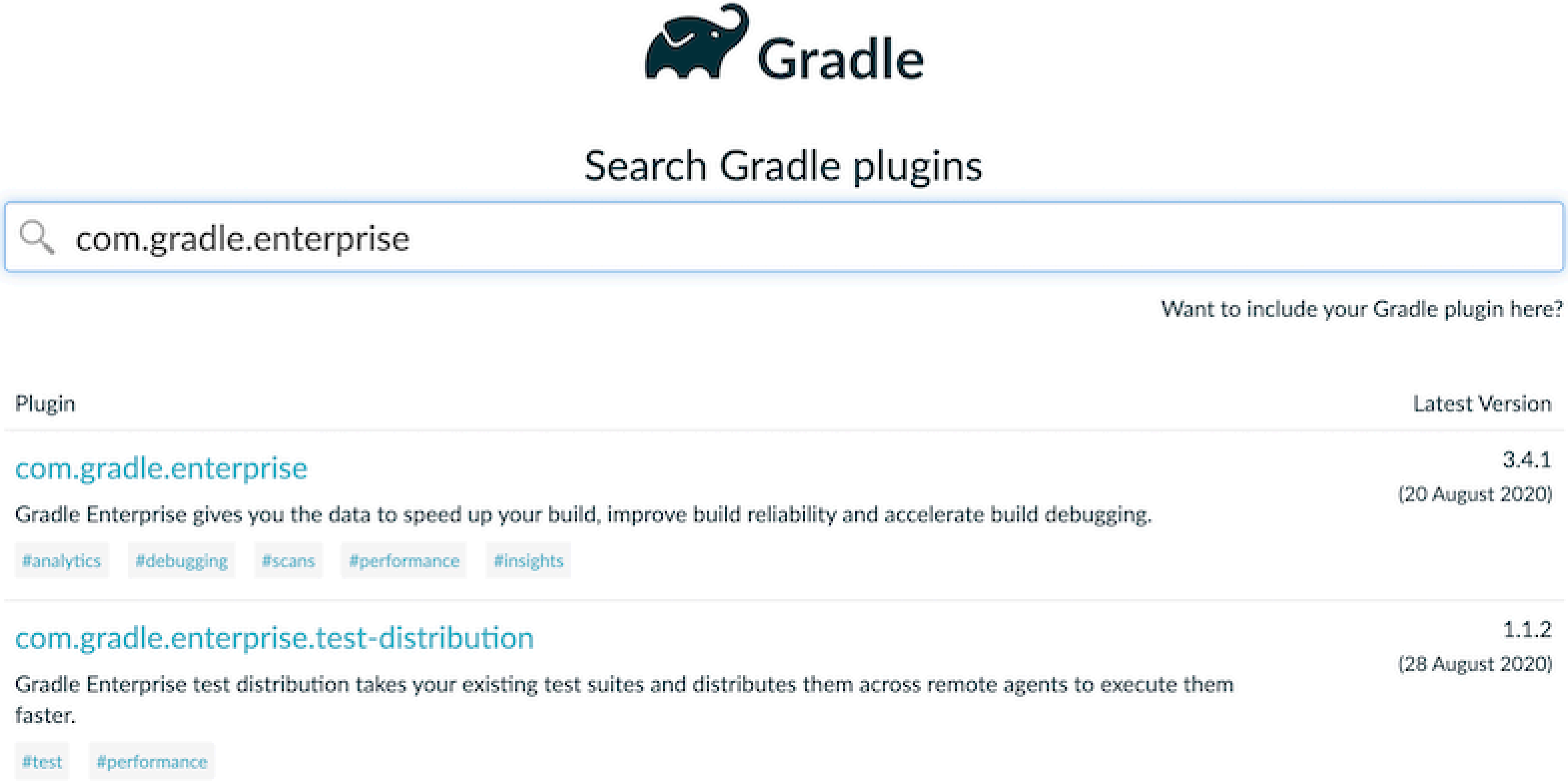
Some examples:

- `java-library` - compile, test, and package Java projects
- `checkstyle` - static analysis for Java code
- `maven-publish` - upload artifacts to Apache Maven repositories
- `scala` - compile, test, package, upload Scala projects
- `application` - support packaging your Java code as a runnable application
- `cpp-library` - support building native binaries using gcc, clang or visual-cpp

Many more, [listed in the Gradle User Manual]().

Gradle Plugin Portal

- Search and discover community plugins on plugins.gradle.org/
- Plugin JARs and their metadata are hosted by Gradle Inc.



Gradle

Search Gradle plugins

com.gradle.enterprise

Want to include your Gradle plugin here?

Plugin	Latest Version
com.gradle.enterprise Gradle Enterprise gives you the data to speed up your build, improve build reliability and accelerate build debugging. #analytics #debugging #scans #performance #insights	3.4.1 (20 August 2020)
com.gradle.enterprise.test-distribution Gradle Enterprise test distribution takes your existing test suites and distributes them across remote agents to execute them faster. #test #performance	1.1.2 (28 August 2020)

Notable community plugins

- Kotlin JVM plugin - Builds Kotlin code
- Spring Boot plugin - Builds Spring boot applications
- Shadow plugin - Builds shaded jars
- Spotbugs plugin - Runs SpotBugs analysis on Java code

Other Gradle Inc plugins

- Gradle Enterprise - Generates build scans.
- Gradle Enterprise Test distribution - Distributes tests across multiple machines.
- Gradle Test Retry - Mitigate flaky tests by retrying tests when they fail.

Thank You!

- Thank you for attending!
- Questions?
- Feedback?
- gradle.org
- gradle.com