

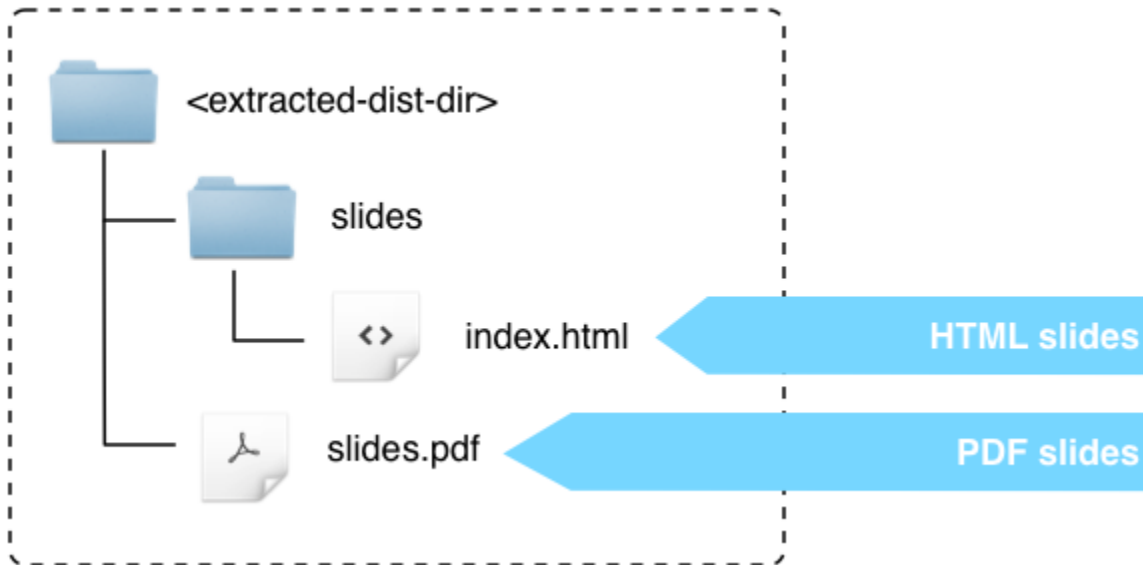
# Introduction to Gradle

The fundamentals of building projects with Gradle

# Slides

---

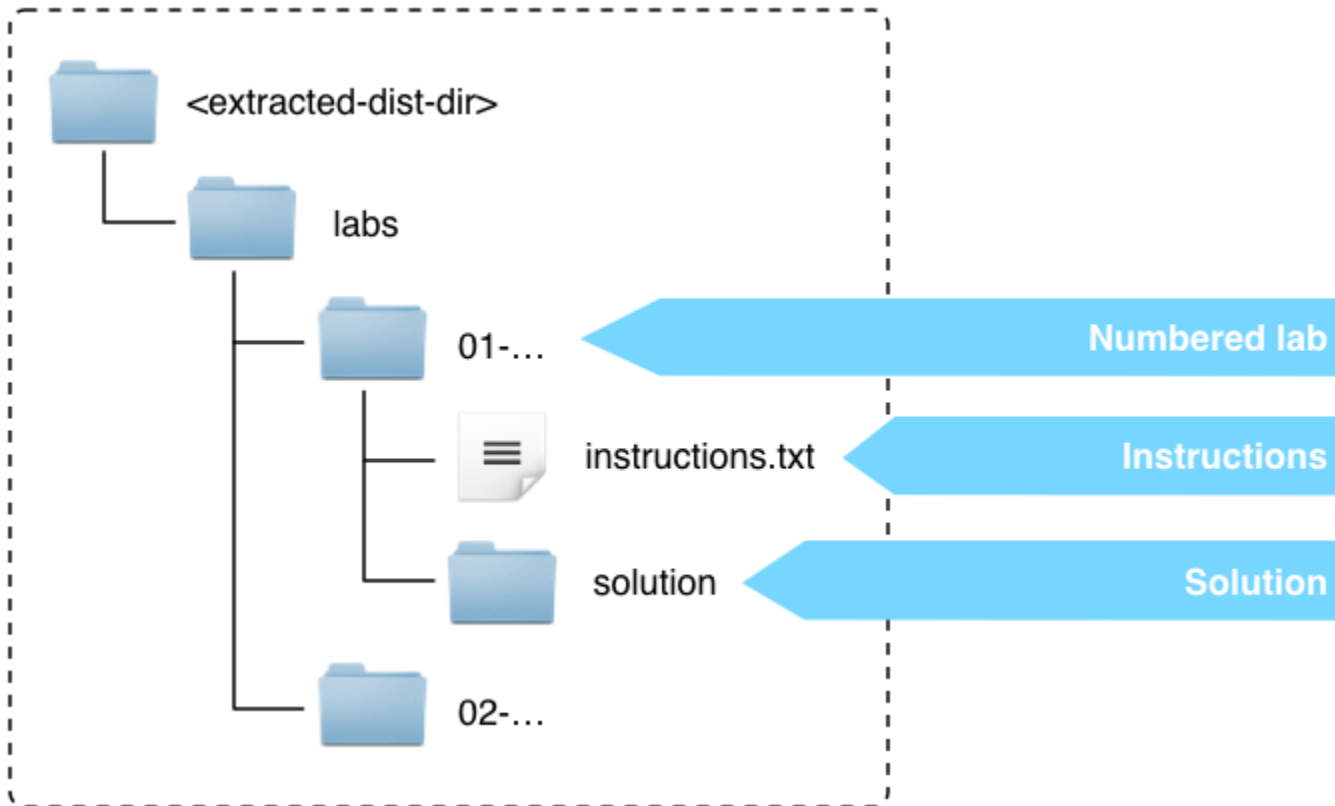
- Available in different formats
- Same content as today's presentation



# Practical labs

---

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



# Ask questions

---

- Please ask questions at any time!
- You control the speed of the presentation
- Q&A session at the end of the workshop

# Objectives

---

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

# Topics we won't cover

---

- Android/Scala builds
- Continuous Integration/Delivery
- IDE integration
- Custom tasks
- Plugin development
- Advanced dependency management techniques

# Agenda

---

- The Gradle Project
- Gradle Basics
- Tasks
- Working with the Filesystem
- Archives
- The Java plugin
- Dependency Management
- Build Scans
- Multi-Project Builds

# Prerequisites

---

1. Ability to read/write Java/Groovy code
2. A basic understanding of build automation



# Gradle

About the project

# Gradle

---

Gradle is a build and automation tool.

Gradle 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 indeed anything else.

- Cross-platform, JVM based
- Implemented in Java



<http://www.gradle.org>

# Gradle Project

---

- Open Source, Apache v2 license - Completely free to use
- Source code on Github - [github.com/gradle/gradle](https://github.com/gradle/gradle)
- Active user community, centered around [discuss.gradle.org](https://discuss.gradle.org)
- Frequent releases (minor releases roughly every 4-8 weeks)
- Strong quality commitment
  - Extensive automated testing (including documentation) - [builds.gradle.org](https://builds.gradle.org)
  - Backwards compatibility & feature lifecycle policy
  - Thorough design and review process
- Developed by domain experts
  - Gradle, Inc. staff and community contributors

# Gradle Documentation

---

- User Guide
  - Many chapters and lots of samples
  - [single page HTML](#), [multi page HTML](#), [PDF](#)
  - Search tip: Full text search on single page HTML
- Build Language Reference ([gradle.org/docs/current/dsl/](https://gradle.org/docs/current/dsl/))
  - Best starting point when authoring build scripts
  - Javadoc-like, but higher-level
  - Links into [Javadoc](#) and [Groovydoc](#)
- Gradle Guides ([guides.gradle.org/](https://guides.gradle.org/))
  - Topic focused (e.g., Plugin Development)
  - Novice to Master

# Other Gradle Resources

---

- Gradle Distribution
  - Includes user guide, build language reference, and **many sample builds**
  - `gradle-all` vs. `gradle-bin`
- [help.gradle.org](https://help.gradle.org)
  - Portal to other helpful resources
- [discuss.gradle.org](https://discuss.gradle.org)
  - Best place to ask questions
- Books, [several are free!](#)

# Running Gradle

# Getting Information

---

Print basic help information:

```
$ gradle help
```

Print command line options:

```
$ gradle -?
```

Print the available tasks in a project:

```
$ gradle tasks
```

# 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
- Use the daemon
  - Gradle 1.x and 2.x have the daemon off by default.
  - Gradle 3.x enables the daemon by default.

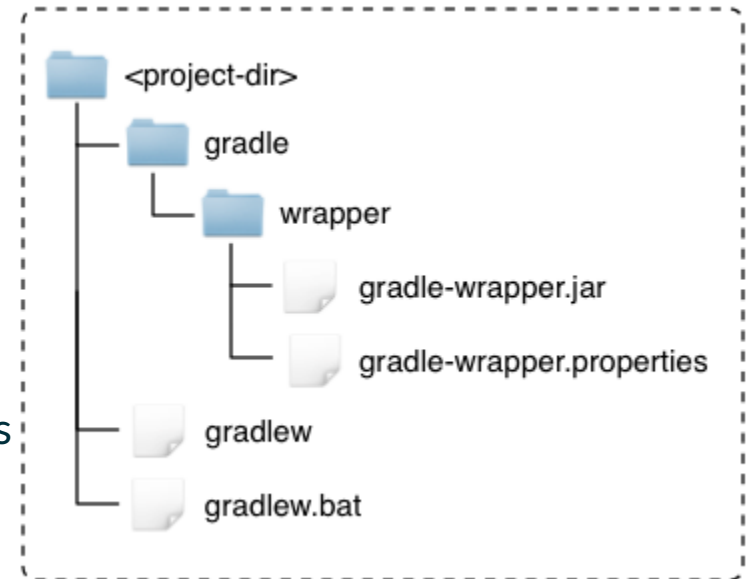


# Gradle Wrapper

# Gradle Wrapper

A way to bootstrap Gradle installations.

- **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



Usage example:

```
$ ./gradlew build
```

# Wrapper task

---

- Wrapper task is built-in and generates:
  - wrapper scripts
  - wrapper jar
  - wrapper properties

```
$ gradle wrapper --gradle-version=4.1
```

- 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 samples.

# Lab

01-wrapper

# Gradle Build Daemon

---

Makes builds start faster. [Way faster.](#)

The daemon is enabled by default since Gradle 3.x.

Use `gradle --status` to see running daemons.

Force shutdown with `gradle --stop`.

- Disable with:
  - `--no-daemon` command line option
  - `org.gradle.daemon=false` in `gradle.properties`
  - `-Dorg.gradle.daemon=false` in `GRADLE_OPTS`

# Lab

02-daemon-setup

# Gradle Basics

# Gradle DSLs

---

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

Kotlin and Groovy bring:

- Domain Specific Language (DSL) capabilities
- Better readability and comprehensibility
- New language features with Java compatibility
- Many useful utilities
- Some useful functional features



# Gradle Build Scripts

---

- Ending in `.gradle.kts`
  - Compiled as Kotlin
- Ending in `.gradle`
  - Compiled as Groovy
- `build.gradle` delegates to `org.gradle.api.Project`
- `settings.gradle` delegates to `org.gradle.api.initialization.Settings`

# Tasks

# Tasks

---

Tasks are the basic unit of work in Gradle.

- Declared & configured in build scripts (or plugins)
- **Executed by Gradle**

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

task is a keyword in the Gradle DSL.

All tasks implement the Task interface.

# Task Actions

---

Tasks have a *list* of actions.

```
task helloWorld {  
    doLast {  
        println "World!"  
    }  
    doFirst {  
        println "Hello"  
    }  
}
```

Most tasks have one useful main action.

`doLast()` and `doFirst()` can be used to decorate the action.

# Executing Tasks

---

Execute tasks by specifying them on the command line.

```
$ gradle helloWorld  
:helloWorld  
Hello world!
```

# Abbrev. Task Name Execution

---

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

```
task myNameIsPrettyLong {  
    doLast {  
        println "Long name"  
    }  
}
```

```
task someOtherTask {  
    doLast {  
        println "Other task"  
    }  
}
```

```
//running:
```

```
$ gradle mNIPL sOtherT
```

Characters between word boundaries can be dropped.

# Lab

03-tasks

# Task types

---

Most of the time, tasks will use an existing task type.

```
task copyFiles(type: Copy) {  
    // configure the task  
}
```

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

---

```
task helloWorld {  
    onlyIf { System.getProperty("be.nice") == "true" }  
    doLast { println "Hello" }  
}
```

The `onlyIf()` and `doLast()` methods are available for all tasks (i.e. part of the `Task` interface).

```
task copyFiles(type: Copy) {  
    from "sourceDir"  
    into "targetDir"  
}
```

The `from()` and `into()` methods are only available for `Copy` tasks.

The task's API allows the task to be *configured*.

# Question

---



When would you implement a task type over an ad-hoc task?

1. never
2. if I want to reuse an existing task definition
3. if the logic becomes complex
4. if I want to implement ad-hoc logic
5. always

# Task Dependencies

---

- Tasks can depend on each other
- Execution of one task requires prior execution of another task
- Executed tasks form a directed acyclic graph

```
task foo

task bar {
    dependsOn foo
}
```

# 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.*

# Question

---



Task actions are executed

1. in the initialization phase
2. in the configuration phase
3. in the execution phase
4. in the last phase of Gradle's build lifecycle

# Lab

04-build-lifecycle

# Bonus Question

---

What will happen here?

```
task bar {  
    doLast {  
        dependsOn foo  
    }  
}
```

Will `foo` execute before `bar`?

# 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 File properties

---

Important properties:

- `projectDir` (read-only) - the base directory of the project
- `buildDir` - the build output directory of the project
- `rootDir` (read-only) - 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.

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

Powerful API, including filtering and transforming.

Used very often in "standard builds".

# Multiple sources/sub directories

---

API has a tree like structure.

```
task copyStuff(type: Copy) {  
    exclude "**/.svn" // default  
    into "targetDir"  
    into("targetSubDir") {  
        from "sourceDir"  
    }  
    into("targetSubDir2") {  
        from "sourceDir2", "someFile.txt"  
    }  
    into("targetSubDir3") {  
        from "sourceDir3"  
        include "**/*.jpeg"  
        exclude "**/obsoleteImages/*"  
    }  
}
```

# Transforming

---

Files can be mutated during copy.

```
task copyStuff(type: 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.

```
task copyStuff(type: 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

05-copy



# Permissions

---

Permissions at the destination can be specified.

```
task copyStuff(type: 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).

```
task ide(type: Sync) {  
    from sharedNetworkLibsDir  
    into "ide"  
}
```

Full copy (not incremental like `rsync`).

# Question

---



## A Copy task can

1. filter files
2. set permissions for files and directories
3. delete files
4. copy directories recursively
5. rename files

# 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

---

```
task zipLibs(type: 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.

```
apply plugin: "base"
version = "1.0"
task zipLibs(type: Zip) {
    baseName = "services"
    appendix = "api"
    // ...
}
```

- baseName -> project.name
- appendix -> empty string
- version -> project.version
- classifier -> empty string
- extension -> type extension

Pattern: «*baseName*»-«*appendix*»-«*version*»-«*classifier*».«*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:

```
apply plugin: "base"

task myZip(type: Zip) {
    destinationDir = file("$buildDir/specialZips")
}
```



# Unarchiving

---

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

```
task unpackArchives(type: 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.

```
task mergedZip(type: Zip) {  
    from zipTree("someZip.zip")  
    from zipTree("otherZip.zip")  
}
```

Typical use case: fat jars.

# Java Plugin

# Java Plugin

---

The basis of Java development with Gradle.

- Introduces concept of source sets
- “main” and “test” source set conventions
- Compilation pre-configured
- Dependency management
- JUnit & TestNG testing
- Produces single JAR
- Javadoc generation
- Standard lifecycle/interface

# Source Sets

---

A logical compilation/processing unit of sources.

- Java source files
- Non compiled source files (e.g. properties files)
- Classpath (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` 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 JUnit 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)

# Lab

06-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:4.0.5.RELEASE"  
    implementation group: "org.springframework", name: "spring-web",  
        version: "4.0.5.RELEASE"  
}
```

Group/Module/Version

# Configurations

---

Dependencies are assigned to *configurations*.

```
configurations {  
    // default with "java" 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 {  
    implementation "org.foo:bar:1.0", {  
        exclude module: "spring-core"  
    }  
}
```

Configuration-wide...

```
configurations {  
    implementation {  
        exclude module: "spring-core"  
    }  
}
```

# 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.

Can use Ivy symbolic versions.

```
dependencies {  
    implementation "org.company:some-lib:2.+"  
    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 "codehaus"  
        url "http://repository.codehaus.org"  
    }  
  
    ivy {  
        url "http://repo.mycompany.com"  
        layout "gradle" // default  
    }  
  
    flatDir(dirs: ["dir1", "dir2"])  
}
```

# Question

---



Dependency management in Gradle can be configured to do the following

1. avoid pulling in transitive dependencies
2. look up dependencies in the local Maven repository
3. download zip files
4. retrieve dependencies from an Ivy repository
5. use files from the local file system as dependencies

# Lab

07-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
- Recommended to use over "old" publishing functionality e.g. [Maven plugin](#)

# Publishing to Maven Repositories

---

```
apply plugin: 'java'
apply plugin: 'maven-publish'

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

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

# Gradle Build Scans

# Creating build scans

---

- Creating a build scan is free.
- Build scans are a permanent, 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 a Gradle, Inc server.** A self-hosted version is available.
- See [Gradle Build Scans](#) for more information.

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

08-create-build-scan



# Multi-Project Builds

# Multi-Project Builds

---

- Flexible directory layout
- Configuration injection
- Project dependencies & partial builds
- Separate configuration/execution hierarchy

# Configuration Injection

---

- **ultimateApp**
  - api
  - services
    - webservice
  - shared

```
subprojects {  
    apply plugin: "java"  
    dependencies {  
        testImplementation "junit:junit:4.7"  
    }  
    test {  
        jvmArgs "-Xmx512M"  
    }  
}
```

# Filtered Injection

---

- ultimateApp
  - api
  - services
    - webservice
  - shared

```
configure(nonWebProjects()) {  
    jar.manifest.attributes Implementor: "Gradle Inc."  
}  
  
def nonWebProjects() {  
    subprojects.findAll { !it.name.startsWith("web") }  
}
```

# Project Dependencies

---

- ultimateApp
  - api
  - services
    - webservice
  - shared

```
dependencies {  
    implementation "commons-lang:commons-lang:2.4"  
    implementation project(":shared")  
}
```

# Partial Builds

---

- ultimateApp
  - api
  - services
    - webservice
  - shared

```
$ gradle build
$ gradle buildDependents
$ gradle buildNeeded
```

# Name Matching Execution

---

- **ultimateApp**
  - api
  - services
    - webservice
  - shared

```
$ gradle build  
$ gradle classes  
$ gradle war
```

# Task/Project Paths

---

- For projects and tasks there is a fully qualified path notation:
  - : (root project)
  - :clean (the clean task of the root project)
  - :api (the api project)
  - :services:webservice (the webservice project)
  - :services:webservice:clean (the clean task of webservice)

```
$ gradle :api:classes
```



# Defining a Multi-Project Build

---

- settings.gradle (location defines the root)
- root project is implicitly included

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

//Everything is configurable:

//default: root dir name
rootProject.name = "main"

//default: "api" dir
project(":api").projectDir = file("/myLocation")

//default: "build.gradle"
project(":shared").buildFileName = "shared.gradle"
```

# Question

---



In a multi-project build, one can

1. define project dependencies
2. inject configuration from the root project to only Java-based subprojects
3. not nest subprojects deeper than one level
4. run a single task from one of the subprojects
5. not change the name of a subproject

# Feature Tour

A selection of useful features not covered so far.

# Init Plugin

---

Create build from template, convert Maven build.

```
$ gradle init --type java-library
```

```
$ gradle init --type pom
```

# Continue after Failure

---

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

Especially useful for CI builds.

# Parallel Builds

---

Run independent tasks from different projects in parallel.

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

Incubating feature; some restrictions apply.

# 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
```

Incubating feature.

Limitations:

- Changes to build scripts will not trigger a build.

# Standard Gradle Plugins

---

Gradle ships with many useful plugins.

Some examples:

- `java` - compile, test, package, upload Java projects
- `checkstyle` - static analysis for Java code
- `maven` - uploading artifacts to Apache Maven repositories
- `scala` - compile, test, package, upload Scala projects
- `idea` and `eclipse` - generates metadata so IDEs understand the project
- `application` - support packaging your Java code as a runnable application
- `c / cpp` - support building native binaries using gcc, clang or visual-cpp

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



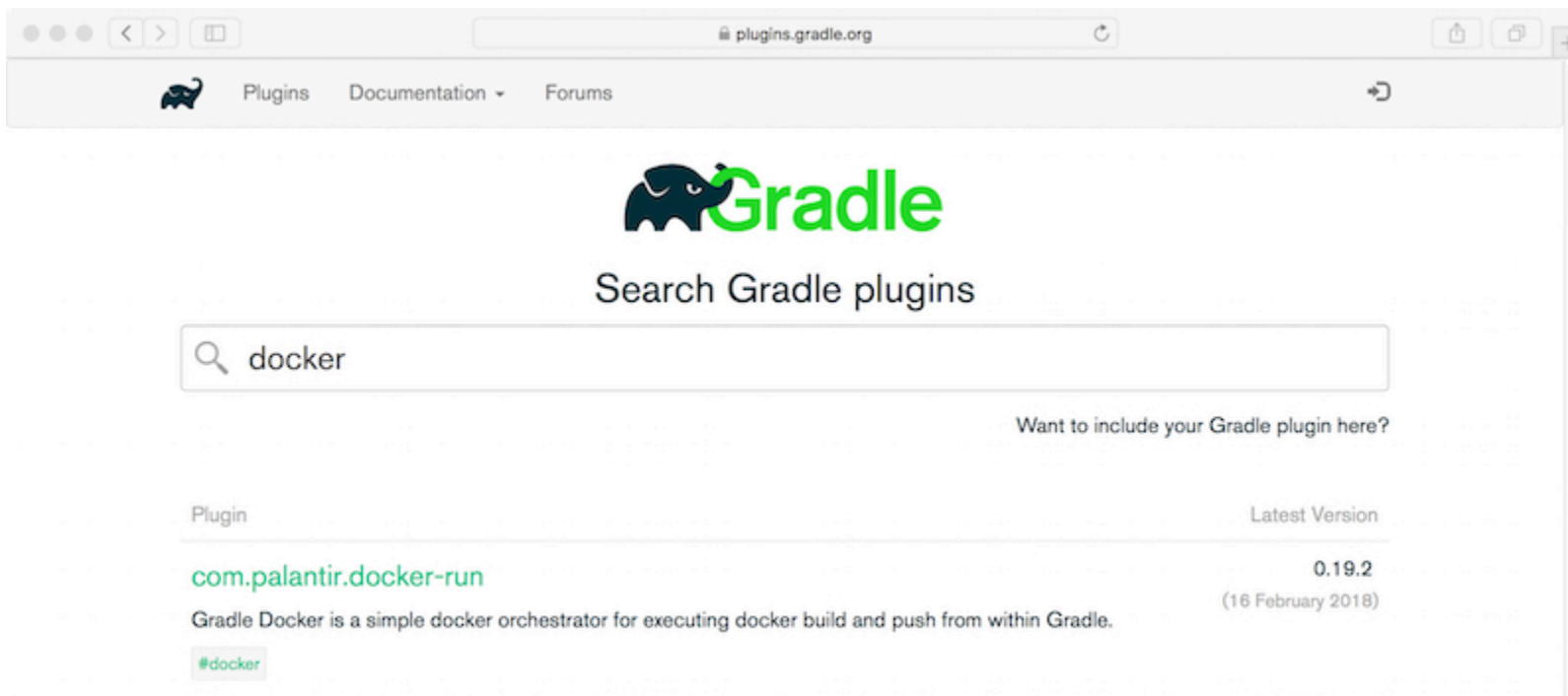
# Notable community plugins

---

- [Android](#): Development of mobile applications for Android
- [Gretty](#): Running web applications on Servlet containers
- [Docker](#): Managing Docker images, containers and work flows
- [SSH](#): Remote command execution and file transfer via SSH
- [Git](#): Executing Git operations, GitHub Pages publishing, Inferring versioning

# Gradle's plugin portal

- Internet accessible plugin portal at <https://plugins.gradle.org/>
- Search and discover community plugins
- Publishing of plugin with dedicated [plugin publishing plugin](#)
- Plugin JARs and their metadata are hosted by Gradle Inc.



# Thank You!

---

- Thank you for attending!
- Questions?
- Feedback?
- [Gradle Home](#)
- [Get more help!](#)