Docs

[User Manual](http://docs.google.com/userguide/userguide.html)

[Guides and Tutorials](https://guides.gradle.org)

[DSL Reference](http://docs.google.com/dsl/)

[Javadoc](http://docs.google.com/javadoc/)

[Release Notes](http://docs.google.com/release-notes.html)

[Forums](https://discuss.gradle.org/)

[Training](https://gradle.org/training/)

[Try Gradle Enterprise](https://gradle.com/enterprise)

[PDF](http://docs.google.com/userguide/userguide.pdf)

* [User Manual Home](http://docs.google.com/userguide/userguide.html)
* [Release Notes](http://docs.google.com/release-notes.html)
* [Installing Gradle](http://docs.google.com/userguide/installation.html)
* [Tutorials](https://guides.gradle.org/)

### Reference

* [Groovy DSL Reference](http://docs.google.com/dsl/)
* [Gradle API Javadoc](http://docs.google.com/javadoc/)
* [Core Plugins](http://docs.google.com/userguide/plugin_reference.html)
* [Gradle & Third-party Tools](http://docs.google.com/userguide/third_party_integration.html)

### Getting Started

* [Creating New Gradle Builds](https://guides.gradle.org/creating-new-gradle-builds/)
* [Creating Build Scans](https://guides.gradle.org/creating-build-scans/)
* [Migrating From Maven](https://guides.gradle.org/migrating-from-maven/)

### Running Gradle Builds

* [Command-Line Interface](http://docs.google.com/userguide/command_line_interface.html)
* [Customizing Execution](#gjdgxs)
  + [Configuring the Build Environment](http://docs.google.com/userguide/build_environment.html)
  + [Configuring the Gradle Daemon](http://docs.google.com/userguide/gradle_daemon.html)
  + [Initialization Scripts](http://docs.google.com/userguide/init_scripts.html)
* [Directory Layout](http://docs.google.com/userguide/directory_layout.html)
* [Executing Multi-Project Builds](http://docs.google.com/userguide/intro_multi_project_builds.html)
* [Gradle Wrapper](http://docs.google.com/userguide/gradle_wrapper.html)
* [Troubleshooting](http://docs.google.com/userguide/troubleshooting.html)
* [Using Build Scans](https://docs.gradle.com/build-scan-plugin)
* [Enabling and Configuring the Build Cache](http://docs.google.com/userguide/build_cache.html)
* [Integrating Separate Gradle Builds (Composite Builds)](http://docs.google.com/userguide/composite_builds.html)

### Authoring Gradle Builds

* [Fundamentals](#30j0zll)
  + [Introducing the Basics of Build Scripts](http://docs.google.com/userguide/tutorial_using_tasks.html)
  + [Working with Tasks](http://docs.google.com/userguide/more_about_tasks.html)
  + [Learning More About Build Scripts](http://docs.google.com/userguide/writing_build_scripts.html)
  + [Working with Files](http://docs.google.com/userguide/working_with_files.html)
  + [Creating Custom Task Types](http://docs.google.com/userguide/custom_tasks.html)
  + [Using Gradle Plugins](http://docs.google.com/userguide/plugins.html)
  + [The Standard Gradle Plugins](http://docs.google.com/userguide/standard_plugins.html)
  + [Understanding the Build Lifecycle](http://docs.google.com/userguide/build_lifecycle.html)
  + [Working with Logging](http://docs.google.com/userguide/logging.html)
  + [Configuring Multi-Project Builds](http://docs.google.com/userguide/multi_project_builds.html)
* [Best Practices](#1fob9te)
  + [Authoring Maintainable Build Scripts](http://docs.google.com/userguide/authoring_maintainable_build_scripts.html)
  + [Organizing Gradle Projects](http://docs.google.com/userguide/organizing_gradle_projects.html)
  + [Optimizing Build Performance](https://guides.gradle.org/performance/)
  + [Using the Build Cache](https://guides.gradle.org/using-build-cache/)
* [Dependency Management](#3znysh7)
  + [Introduction to Dependency Management](http://docs.google.com/userguide/introduction_dependency_management.html)
  + [Dependency Management Terminology](http://docs.google.com/userguide/dependency_management_terminology.html)
  + [Dependency Types](http://docs.google.com/userguide/dependency_types.html)
  + [Repository Types](http://docs.google.com/userguide/repository_types.html)
  + [Declaring Dependencies](http://docs.google.com/userguide/declaring_dependencies.html)
  + [Declaring Repositories](http://docs.google.com/userguide/declaring_repositories.html)
  + [Inspecting Dependencies](http://docs.google.com/userguide/inspecting_dependencies.html)
  + [Managing Dependency Configurations](http://docs.google.com/userguide/managing_dependency_configurations.html)
  + [Managing Transitive Dependencies](http://docs.google.com/userguide/managing_transitive_dependencies.html)
  + [Dependency Locking](http://docs.google.com/userguide/dependency_locking.html)
  + [Troubleshooting Dependency Resolution](http://docs.google.com/userguide/troubleshooting_dependency_resolution.html)
  + [Customizing Dependency Resolution Behavior](http://docs.google.com/userguide/customizing_dependency_resolution_behavior.html)
  + [Dependency Cache Internals](http://docs.google.com/userguide/dependency_cache.html)
  + [Working with Dependencies](http://docs.google.com/userguide/working_with_dependencies.html)
* [Publishing Artifacts](http://docs.google.com/userguide/artifact_management.html)
* [C++ Projects](#2et92p0)
  + [Building Native Software](http://docs.google.com/userguide/native_software.html)
  + [Software Model Concepts](http://docs.google.com/userguide/software_model_concepts.html)
  + [Rule-based Model Configuration](http://docs.google.com/userguide/software_model.html)
  + [Implementing Model Rules in a Plugin](http://docs.google.com/userguide/rule_source.html)
  + [Extending the Software Model](http://docs.google.com/userguide/software_model_extend.html)
* [Java Projects](#tyjcwt)
  + [Building Java & JVM projects](http://docs.google.com/userguide/building_java_projects.html)
  + [Testing Java & JVM projects](http://docs.google.com/userguide/java_testing.html)
* [Advanced Techniques](#3dy6vkm)
  + [Configuring Tasks Lazily](http://docs.google.com/userguide/lazy_configuration.html)
  + [Developing Parallel Tasks](https://guides.gradle.org/using-the-worker-api/)
  + [Testing Your Build with TestKit](http://docs.google.com/userguide/test_kit.html)
  + [Using Ant from Gradle](http://docs.google.com/userguide/ant.html)
* [Sample Gradle builds](#1t3h5sf)
  + [Groovy DSL Samples](https://github.com/gradle/gradle/tree/master/subprojects/docs/src/samples)
  + [Kotlin DSL Samples](https://github.com/gradle/kotlin-dsl/tree/master/samples)

### Extending Gradle

* [Writing Custom Plugins](http://docs.google.com/userguide/custom_plugins.html)
* [Plugin Development Guides](https://gradle.org/guides/?q=Plugin+Development)

[Edit this page](https://github.com/gradle/gradle/edit/master/subprojects/docs/src/docs/userguide/)

# Authoring Multi-Project Builds

Contents

[Cross project configuration](#4d34og8)

[Subproject configuration](#2s8eyo1)

[Execution rules for multi-project builds](#17dp8vu)

[Running tasks by their absolute path](#3rdcrjn)

[Project and task paths](#26in1rg)

[Dependencies - Which dependencies?](#lnxbz9)

[Project lib dependencies](#35nkun2)

[Depending on the task output produced by another project](#1ksv4uv)

[Parallel project execution](#44sinio)

[Decoupled Projects](#2jxsxqh)

[Multi-Project Building and Testing](#z337ya)

[Multi Project and buildSrc](#3j2qqm3)

The powerful support for multi-project builds is one of Gradle’s unique selling points. This topic is also the most intellectually challenging.

A multi-project build in gradle consists of one root project, and one or more subprojects that may also have subprojects.

[Cross project configuration](#4d34og8)

While each subproject could configure itself in complete isolation of the other subprojects, it is common that subprojects share common traits. It is then usually preferable to share configurations among projects, so the same configuration affects several subprojects.

Let’s start with a very simple multi-project build. Gradle is a general purpose build tool at its core, so the projects don’t have to be Java projects. Our first examples are about marine life.

[Configuration and execution](#1y810tw)

[Build phases](http://docs.google.com/build_lifecycle.html#sec:build_phases) describes the phases of every Gradle build. Let’s zoom into the configuration and execution phases of a multi-project build. Configuration here means executing the build.gradle file of a project, which implies e.g. downloading all plugins that were declared using ‘apply plugin’. By default, the configuration of all projects happens before any task is executed. This means that when a single task, from a single project is requested, *all* projects of multi-project build are configured first. The reason every project needs to be configured is to support the flexibility of accessing and changing any part of the Gradle project model.

[Configuration on demand](#4i7ojhp)

The *Configuration injection* feature and access to the complete project model are possible because every project is configured before the execution phase. Yet, this approach may not be the most efficient in a very large multi-project build. There are Gradle builds with a hierarchy of hundreds of subprojects. The configuration time of huge multi-project builds may become noticeable. Scalability is an important requirement for Gradle. Hence, starting from version 1.4 a new incubating 'configuration on demand' mode is introduced.

Configuration on demand mode attempts to configure only projects that are relevant for requested tasks, i.e. it only executes the build.gradle file of projects that are participating in the build. This way, the configuration time of a large multi-project build can be reduced. In the long term, this mode will become the default mode, possibly the only mode for Gradle build execution. The configuration on demand feature is incubating so not every build is guaranteed to work correctly. The feature should work very well for multi-project builds that have [decoupled projects](#2jxsxqh). In “configuration on demand” mode, projects are configured as follows:

* The root project is always configured. This way the typical common configuration is supported (allprojects or subprojects script blocks).
* The project in the directory where the build is executed is also configured, but only when Gradle is executed without any tasks. This way the default tasks behave correctly when projects are configured on demand.
* The standard project dependencies are supported and makes relevant projects configured. If project A has a compile dependency on project B then building A causes configuration of both projects.
* The task dependencies declared via task path are supported and cause relevant projects to be configured. Example: someTask.dependsOn(":someOtherProject:someOtherTask")
* A task requested via task path from the command line (or Tooling API) causes the relevant project to be configured. For example, building 'projectA:projectB:someTask' causes configuration of projectB.

Eager to try out this new feature? To configure on demand with every build run see [Gradle properties](http://docs.google.com/build_environment.html#sec:gradle_configuration_properties). To configure on demand just for a given build, see [command-line performance-oriented options](http://docs.google.com/command_line_interface.html#sec:command_line_performance).

[Defining common behavior](#2xcytpi)

Let’s look at some examples with the following project tree. This is a multi-project build with a root project named water and a subproject named bluewhale.

[Example: Multi-project tree - water & bluewhale projects](#1ci93xb)

**Project layout**

.  
├── bluewhale/  
├── build.gradle  
└── settings.gradle

| **✨** | The code for this example can be found at samples/userguide/multiproject/firstExample/water in the ‘-all’ distribution of Gradle. |
| --- | --- |

**settings.gradle**

rootProject.name = 'water'  
include 'bluewhale'

And where is the build script for the bluewhale project? In Gradle build scripts are optional. Obviously for a single project build, a project without a build script doesn’t make much sense. For multiproject builds the situation is different. Let’s look at the build script for the water project and execute it:

[Example: Build script of water (parent) project](#3whwml4)

**build.gradle**

Closure cl = { task -> println "I'm $task.project.name" }  
task('hello').doLast(cl)  
project(':bluewhale') {  
 task('hello').doLast(cl)  
}

**Output of** gradle -q hello

> gradle -q hello  
I'm water  
I'm bluewhale

Gradle allows you to access any project of the multi-project build from any build script. The Project API provides a method called project(), which takes a path as an argument and returns the Project object for this path. The capability to configure a project build from any build script we call *cross project configuration*. Gradle implements this via *configuration injection*.

We are not that happy with the build script of the water project. It is inconvenient to add the task explicitly for every project. We can do better. Let’s first add another project called krill to our multi-project build.

[Example: Multi-project tree - water, bluewhale & krill projects](#2bn6wsx)

**Project layout**

.  
├── bluewhale/  
├── build.gradle  
├── krill/  
└── settings.gradle

| **✨** | The code for this example can be found at samples/userguide/multiproject/addKrill/water in the ‘-all’ distribution of Gradle. |
| --- | --- |

**settings.gradle**

rootProject.name = 'water'  
  
include 'bluewhale', 'krill'

Now we rewrite the water build script and boil it down to a single line.

[Example: Water project build script](#qsh70q)

**build.gradle**

allprojects {  
 task hello {  
 doLast { task ->  
 println "I'm $task.project.name"  
 }  
 }  
}

**Output of** gradle -q hello

> gradle -q hello  
I'm water  
I'm bluewhale  
I'm krill

Is this cool or is this cool? And how does this work? The Project API provides a property allprojects which returns a list with the current project and all its subprojects underneath it. If you call allprojects with a closure, the statements of the closure are delegated to the projects associated with allprojects. You could also do an iteration via allprojects.each, but that would be more verbose.

Other build systems use inheritance as the primary means for defining common behavior. We also offer inheritance for projects as you will see later. But Gradle uses configuration injection as the usual way of defining common behavior. We think it provides a very powerful and flexible way of configuring multiproject builds.

Another possibility for sharing configuration is to use a common external script.

[Subproject configuration](#2s8eyo1)

The Project API also provides a property for accessing the subprojects only.

[Defining common behavior](#3as4poj)

[Example: Defining common behavior of all projects and subprojects](#1pxezwc)

**build.gradle**

allprojects {  
 task hello {  
 doLast { task ->  
 println "I'm $task.project.name"  
 }  
 }  
}  
subprojects {  
 hello {  
 doLast {  
 println "- I depend on water"  
 }  
 }  
}

**Output of** gradle -q hello

> gradle -q hello  
I'm water  
I'm bluewhale  
- I depend on water  
I'm krill  
- I depend on water

You may notice that there are two code snippets referencing the “hello” task. The first one, which uses the “task” keyword, constructs the task and provides it’s base configuration. The second piece doesn’t use the “task” keyword, as it is further configuring the existing “hello” task. You may only construct a task once in a project, but you may add any number of code blocks providing additional configuration.

[Adding specific behavior](#49x2ik5)

You can add specific behavior on top of the common behavior. Usually we put the project specific behavior in the build script of the project where we want to apply this specific behavior. But as we have already seen, we don’t have to do it this way. We could add project specific behavior for the bluewhale project like this:

[Example: Defining specific behaviour for particular project](#2p2csry)

**build.gradle**

allprojects {  
 task hello {  
 doLast { task ->  
 println "I'm $task.project.name"  
 }  
 }  
}  
subprojects {  
 hello {  
 doLast {  
 println "- I depend on water"  
 }  
 }  
}  
project(':bluewhale').hello {  
 doLast {  
 println "- I'm the largest animal that has ever lived on this planet."  
 }  
}

**Output of** gradle -q hello

> gradle -q hello  
I'm water  
I'm bluewhale  
- I depend on water  
- I'm the largest animal that has ever lived on this planet.  
I'm krill  
- I depend on water

As we have said, we usually prefer to put project specific behavior into the build script of this project. Let’s refactor and also add some project specific behavior to the krill project.

[Example: Defining specific behaviour for project krill](#147n2zr)

**Project layout**

.  
├── bluewhale  
│   └── build.gradle  
├── build.gradle  
├── krill  
│   └── build.gradle  
└── settings.gradle

| **✨** | The code for this example can be found at samples/userguide/multiproject/spreadSpecifics/water in the ‘-all’ distribution of Gradle. |
| --- | --- |

**settings.gradle**

rootProject.name = 'water'  
include 'bluewhale', 'krill'

**bluewhale/build.gradle**

hello.doLast {  
 println "- I'm the largest animal that has ever lived on this planet."  
}

**krill/build.gradle**

hello.doLast {  
 println "- The weight of my species in summer is twice as heavy as all human beings."  
}

**build.gradle**

allprojects {  
 task hello {  
 doLast { task ->  
 println "I'm $task.project.name"  
 }  
 }  
}  
subprojects {  
 hello {  
 doLast {  
 println "- I depend on water"  
 }  
 }  
}

**Output of gradle -q hello**

> gradle -q hello  
I'm water  
I'm bluewhale  
- I depend on water  
- I'm the largest animal that has ever lived on this planet.  
I'm krill  
- I depend on water  
- The weight of my species in summer is twice as heavy as all human beings.

[Project filtering](#3o7alnk)

To show more of the power of configuration injection, let’s add another project called tropicalFish and add more behavior to the build via the build script of the water project.

[Filtering by name](#23ckvvd)

[Example: Adding custom behaviour to some projects (filtered by project name)](#ihv636)

**Project layout**

.  
├── bluewhale/  
│   └── build.gradle  
├── build.gradle  
├── krill/  
│   └── build.gradle  
├── settings.gradle  
└── tropicalFish/

| **✨** | The code for this example can be found at samples/userguide/multiproject/addTropical/water in the ‘-all’ distribution of Gradle. |
| --- | --- |

**settings.gradle**

rootProject.name = 'water'  
include 'bluewhale', 'krill', 'tropicalFish'

**build.gradle**

allprojects {  
 task hello {  
 doLast { task ->  
 println "I'm $task.project.name"  
 }  
 }  
}  
subprojects {  
 hello {  
 doLast {  
 println "- I depend on water"  
 }  
 }  
}  
configure(subprojects.findAll {it.name != 'tropicalFish'}) {  
 hello {  
 doLast {  
 println '- I love to spend time in the arctic waters.'  
 }  
 }  
}

**Output of gradle -q hello**

> gradle -q hello  
I'm water  
I'm bluewhale  
- I depend on water  
- I love to spend time in the arctic waters.  
- I'm the largest animal that has ever lived on this planet.  
I'm krill  
- I depend on water  
- I love to spend time in the arctic waters.  
- The weight of my species in summer is twice as heavy as all human beings.  
I'm tropicalFish  
- I depend on water

The configure() method takes a list as an argument and applies the configuration to the projects in this list.

[Filtering by properties](#32hioqz)

Using the project name for filtering is one option. Using [extra project properties](http://docs.google.com/writing_build_scripts.html#sec:extra_properties) is another. for more information on extra properties.)

[Example: Adding custom behaviour to some projects (filtered by project properties)](#1hmsyys)

**Project layout**

.  
├── bluewhale  
│   └── build.gradle  
├── build.gradle  
├── krill  
│   └── build.gradle  
├── settings.gradle  
└── tropicalFish  
 └── build.gradle

| **✨** | The code for this example can be found at samples/userguide/multiproject/tropicalWithProperties/water in the ‘-all’ distribution of Gradle. |
| --- | --- |

**settings.gradle**

rootProject.name = 'water'  
include 'bluewhale', 'krill', 'tropicalFish'

**bluewhale/build.gradle**

ext.arctic = true  
hello.doLast {  
 println "- I'm the largest animal that has ever lived on this planet."  
}

**krill/build.gradle**

ext.arctic = true  
hello.doLast {  
 println "- The weight of my species in summer is twice as heavy as all human beings."  
}

**build.gradle**

allprojects {  
 task hello {  
 doLast { task ->  
 println "I'm $task.project.name"  
 }  
 }  
}  
subprojects {  
 hello {  
 doLast {println "- I depend on water"}  
 afterEvaluate { Project project ->  
 if (project.arctic) { doLast {  
 println '- I love to spend time in the arctic waters.' }  
 }  
 }  
 }  
}

**tropicalFish/build.gradle**

ext.arctic = false

**Output of gradle -q hello**

> gradle -q hello  
I'm water  
I'm bluewhale  
- I depend on water  
- I'm the largest animal that has ever lived on this planet.  
- I love to spend time in the arctic waters.  
I'm krill  
- I depend on water  
- The weight of my species in summer is twice as heavy as all human beings.  
- I love to spend time in the arctic waters.  
I'm tropicalFish  
- I depend on water

In the build file of the water project we use an afterEvaluate notification. This means that the closure we are passing gets evaluated *after* the build scripts of the subproject are evaluated. As the property arctic is set in those build scripts, we have to do it this way. You will find more on this topic in [Dependencies — Which Dependencies?](#lnxbz9)

[Execution rules for multi-project builds](#17dp8vu)

When we executed the hello task from the root project dir, things behaved in an intuitive way. All the hello tasks of the different projects were executed. Let’s switch to the bluewhale dir and see what happens if we execute Gradle from there.

[Example: Running build from subproject](#41mghml)

**Output of** gradle -q hello

> gradle -q hello  
I'm bluewhale  
- I depend on water  
- I'm the largest animal that has ever lived on this planet.  
- I love to spend time in the arctic waters.

The basic rule behind Gradle’s behavior is simple. Gradle looks down the hierarchy, starting with the *current dir*, for tasks with the name hello and executes them. One thing is very important to note. Gradle *always* evaluates *every* project of the multi-project build and creates all existing task objects. Then, according to the task name arguments and the current dir, Gradle filters the tasks which should be executed. Because of Gradle’s cross project configuration *every* project has to be evaluated before *any* task gets executed. We will have a closer look at this in the next section. Let’s now have our last marine example. Let’s add a task to bluewhale and krill.

[Example: Evaluation and execution of projects](#2grqrue)

**bluewhale/build.gradle**

ext.arctic = true  
hello {  
 doLast {  
 println "- I'm the largest animal that has ever lived on this planet."  
 }  
}  
  
task distanceToIceberg {  
 doLast {  
 println '20 nautical miles'  
 }  
}

**krill/build.gradle**

ext.arctic = true  
hello {  
 doLast {  
 println "- The weight of my species in summer is twice as heavy as all human beings."  
 }  
}  
  
task distanceToIceberg {  
 doLast {  
 println '5 nautical miles'  
 }  
}

**Output of gradle -q distanceToIceberg**

> gradle -q distanceToIceberg  
20 nautical miles  
5 nautical miles

Here’s the output without the -q option:

[Example: Evaluation and execution of projects](#vx1227)

**Output of** gradle distanceToIceberg

> gradle distanceToIceberg  
  
> Task :bluewhale:distanceToIceberg  
20 nautical miles  
  
> Task :krill:distanceToIceberg  
5 nautical miles  
  
BUILD SUCCESSFUL in 0s  
2 actionable tasks: 2 executed

The build is executed from the water project. Neither water nor tropicalFish have a task with the name distanceToIceberg. Gradle does not care. The simple rule mentioned already above is: Execute all tasks down the hierarchy which have this name. Only complain if there is *no* such task!

[Running tasks by their absolute path](#3rdcrjn)

As we have seen, you can run a multi-project build by entering any subproject dir and execute the build from there. All matching task names of the project hierarchy starting with the current dir are executed. But Gradle also offers to execute tasks by their absolute path (see also [Project and task paths](#26in1rg)):

[Example: Running tasks by their absolute path](#3fwokq0)

**Output of** gradle -q :hello :krill:hello hello

> gradle -q :hello :krill:hello hello  
I'm water  
I'm krill  
- I depend on water  
- The weight of my species in summer is twice as heavy as all human beings.  
- I love to spend time in the arctic waters.  
I'm tropicalFish  
- I depend on water

The build is executed from the tropicalFish project. We execute the hello tasks of the water, the krill and the tropicalFish project. The first two tasks are specified by their absolute path, the last task is executed using the name matching mechanism described above.

[Project and task paths](#26in1rg)

A project path has the following pattern: It starts with an optional colon, which denotes the root project. The root project is the only project in a path that is not specified by its name. The rest of a project path is a colon-separated sequence of project names, where the next project is a subproject of the previous project.

The path of a task is simply its project path plus the task name, like “:bluewhale:hello”. Within a project you can address a task of the same project just by its name. This is interpreted as a relative path.

[Dependencies - Which dependencies?](#lnxbz9)

The examples from the last section were special, as the projects had no *Execution Dependencies*. They had only *Configuration Dependencies*. The following sections illustrate the differences between these two types of dependencies.

[Execution dependencies](#1v1yuxt)

[Dependencies and execution order](#4f1mdlm)

[Example: Dependencies and execution order](#2u6wntf)

**Project layout**

.  
├── build.gradle  
├── consumer  
│   └── build.gradle  
├── producer  
│   └── build.gradle  
└── settings.gradle

| **✨** | The code for this example can be found at samples/userguide/multiproject/dependencies/firstMessages/messages in the ‘-all’ distribution of Gradle. |
| --- | --- |

**build.gradle**

ext.producerMessage = null

**settings.gradle**

include 'consumer', 'producer'

**consumer/build.gradle**

task action {  
 doLast {  
 println("Consuming message: ${rootProject.producerMessage}")  
 }  
}

**producer/build.gradle**

task action {  
 doLast {  
 println "Producing message:"  
 rootProject.producerMessage = 'Watch the order of execution.'  
 }  
}

**Output of gradle -q action**

> gradle -q action  
Consuming message: null  
Producing message:

This didn’t quite do what we want. If nothing else is defined, Gradle executes the task in alphanumeric order. Therefore, Gradle will execute “:consumer:action” before “:producer:action”. Let’s try to solve this with a hack and rename the producer project to “aProducer”.

[Example: Dependencies and execution order](#19c6y18)

**Project layout**

.  
├── aProducer  
│   └── build.gradle  
├── build.gradle  
├── consumer  
│   └── build.gradle  
└── settings.gradle

**build.gradle**

ext.producerMessage = null

**settings.gradle**

include 'consumer', 'aProducer'

**aProducer/build.gradle**

task action {  
 doLast {  
 println "Producing message:"  
 rootProject.producerMessage = 'Watch the order of execution.'  
 }  
}

**consumer/build.gradle**

task action {  
 doLast {  
 println("Consuming message: ${rootProject.producerMessage}")  
 }  
}

**Output of gradle -q action**

> gradle -q action  
Producing message:  
Consuming message: Watch the order of execution.

We can show where this hack doesn’t work if we now switch to the consumer dir and execute the build.

[Example: Dependencies and execution order](#3tbugp1)

**Output of** gradle -q action

> gradle -q action  
Consuming message: null

The problem is that the two “action” tasks are unrelated. If you execute the build from the “messages” project Gradle executes them both because they have the same name and they are down the hierarchy. In the last example only one “action” task was down the hierarchy and therefore it was the only task that was executed. We need something better than this hack.

[Real life examples](#28h4qwu)

Gradle’s multi-project features are driven by real life use cases. One good example consists of two web application projects and a parent project that creates a distribution including the two web applications.[[1](#nmf14n)] For the example we use only one build script and do *cross project configuration*.

[Example: Dependencies - real life example - crossproject configuration](#37m2jsg)

**Project layout**

.  
├── build.gradle  
├── date  
│   └── src  
│   └── main  
│   ├── java  
│   │   └── org  
│   │   └── gradle  
│   │   └── sample  
│   │   └── DateServlet.java  
│   └── webapp  
│   └── web.xml  
├── hello  
│   └── src  
│   └── main  
│   ├── java  
│   │   └── org  
│   │   └── gradle  
│   │   └── sample  
│   │   └── HelloServlet.java  
│   └── webapp  
│   └── web.xml  
└── settings.gradle

| **✨** | The code for this example can be found at samples/userguide/multiproject/dependencies/webDist in the ‘-all’ distribution of Gradle. |
| --- | --- |

**settings.gradle**

include 'date', 'hello'

**build.gradle**

allprojects {  
 apply plugin: 'java'  
 group = 'org.gradle.sample'  
 version = '1.0'  
}  
  
subprojects {  
 apply plugin: 'war'  
 repositories {  
 mavenCentral()  
 }  
 dependencies {  
 compile "javax.servlet:servlet-api:2.5"  
 }  
}  
  
task explodedDist(type: Copy) {  
 into "$buildDir/explodedDist"  
 subprojects {  
 from tasks.withType(War)  
 }  
}

We have an interesting set of dependencies. Obviously the date and hello projects have a *configuration* dependency on webDist, as all the build logic for the webapp projects is injected by webDist. The *execution* dependency is in the other direction, as webDist depends on the build artifacts of date and hello. There is even a third dependency. webDist has a *configuration* dependency on date and hello because it needs to know the archivePath. But it asks for this information at *execution time*. Therefore we have no circular dependency.

Such dependency patterns are daily bread in the problem space of multi-project builds. If a build system does not support these patterns, you either can’t solve your problem or you need to do ugly hacks which are hard to maintain and massively impair your productivity as a build master.

[Project lib dependencies](#35nkun2)

What if one project needs the jar produced by another project in its compile path, and not just the jar but also the transitive dependencies of this jar? Obviously this is a very common use case for Java multi-project builds. As mentioned in [Project dependencies](http://docs.google.com/dependency_types.html#sub:project_dependencies), Gradle offers project lib dependencies for this.

**Project lib dependencies**

.  
├── api  
│   └── src  
│   ├── main  
│   │   └── java  
│   │   └── org  
│   │   └── gradle  
│   │   └── sample  
│   │   ├── api  
│   │   │   └── Person.java  
│   │   └── apiImpl  
│   │   └── PersonImpl.java  
│   └── test  
│   └── java  
│   └── org  
│   └── gradle  
│   └── PersonTest.java  
├── build.gradle  
├── services  
│   └── personService  
│   └── src  
│   ├── main  
│   │   └── java  
│   │   └── org  
│   │   └── gradle  
│   │   └── sample  
│   │   └── services  
│   │   └── PersonService.java  
│   └── test  
│   └── java  
│   └── org  
│   └── gradle  
│   └── sample  
│   └── services  
│   └── PersonServiceTest.java  
├── settings.gradle  
└── shared  
 └── src  
 └── main  
 └── java  
 └── org  
 └── gradle  
 └── sample  
 └── shared  
 └── Helper.java

| **✨** | The code for this example can be found at samples/userguide/multiproject/dependencies/java in the ‘-all’ distribution of Gradle. |
| --- | --- |

We have the projects “shared”, “api” and “personService”. The “personService” project has a lib dependency on the other two projects. The “api” project has a lib dependency on the “shared” project. “services” is also a project, but we use it just as a container. It has no build script and gets nothing injected by another build script. We use the : separator to define a project path. Consult the DSL documentation of [Settings.include(java.lang.String[])](http://docs.google.com/dsl/org.gradle.api.initialization.Settings.html#org.gradle.api.initialization.Settings:include(java.lang.String%5B%5D)) for more information about defining project paths.

[Example: Project lib dependencies](#1mrcu09)

**settings.gradle**

include 'api', 'shared', 'services:personService'

**build.gradle**

subprojects {  
 apply plugin: 'java'  
 group = 'org.gradle.sample'  
 version = '1.0'  
 repositories {  
 mavenCentral()  
 }  
 dependencies {  
 testCompile "junit:junit:4.12"  
 }  
}  
  
project(':api') {  
 dependencies {  
 compile project(':shared')  
 }  
}  
  
project(':services:personService') {  
 dependencies {  
 compile project(':shared'), project(':api')  
 }  
}

All the build logic is in the “build.gradle” file of the root project.[[2](#46r0co2)] A “*lib*” dependency is a special form of an execution dependency. It causes the other project to be built first and adds the jar with the classes of the other project to the classpath. It also adds the dependencies of the other project to the classpath. So you can enter the “api” directory and trigger a “gradle compile”. First the “shared” project is built and then the “api” project is built. Project dependencies enable partial multi-project builds.

If you come from Maven land you might be perfectly happy with this. If you come from Ivy land, you might expect some more fine grained control. Gradle offers this to you:

[Example: Fine grained control over dependencies](#2lwamvv)

**build.gradle**

subprojects {  
 apply plugin: 'java'  
 group = 'org.gradle.sample'  
 version = '1.0'  
}  
  
project(':api') {  
 configurations {  
 spi  
 }  
 dependencies {  
 compile project(':shared')  
 }  
 task spiJar(type: Jar) {  
 baseName = 'api-spi'  
 from sourceSets.main.output  
 include('org/gradle/sample/api/\*\*')  
 }  
 artifacts {  
 spi spiJar  
 }  
}  
  
project(':services:personService') {  
 dependencies {  
 compile project(':shared')  
 compile project(path: ':api', configuration: 'spi')  
 testCompile "junit:junit:4.12", project(':api')  
 }  
}

The Java plugin adds per default a jar to your project libraries which contains all the classes. In this example we create an *additional* library containing only the interfaces of the “api” project. We assign this library to a new *dependency configuration*. For the person service we declare that the project should be compiled only against the “api” interfaces but tested with all classes from “api”.

[Depending on the task output produced by another project](#1ksv4uv)

[Project dependencies](#35nkun2) model dependencies between modules. Effectively, you are saying that you depend on the main output of another project. In a Java-based project that’s usually a JAR file.

Sometimes you may want to depend on an output produced by another task. In turn you’ll want to make sure that the task is executed beforehand to produce that very output. Declaring a task dependency from one project to another is a poor way to model this kind of relationship and introduces unnecessary coupling. The recommended way to model such a dependency is to produce the output, mark it as an "outgoing" artifact or add it to the output of the main source set which you can depend on in the consuming project.

Let’s say you are working in a multi-project build with the two subprojects producer and consumer. The subproject producer defines a task named buildInfo that generates a properties file containing build information e.g. the project version. The attribute builtBy takes care of establishing an inferred task dependency. For more information on builtBy, see [SourceSetOutput](http://docs.google.com/dsl/org.gradle.api.tasks.SourceSetOutput.html).

[Example: Task generating a property file containing build information](#111kx3o)

**build.gradle**

task buildInfo(type: BuildInfo) {  
 version = project.version  
 outputFile = file("${buildDir}/generated-resources/build-info.properties")  
}  
  
sourceSets {  
 main {  
 output.dir(buildInfo.outputFile.parentFile, builtBy: buildInfo)  
 }  
}

The consuming project is supposed to be able to read the properties file at runtime. Declaring a project dependency on the producing project takes care of creating the properties beforehand and making it available to the runtime classpath.

[Example: Declaring a project dependency on the project producing the properties file](#3l18frh)

**build.gradle**

dependencies {  
 runtime project(':producer')  
}

In the example above, the consumer now declares a dependency on the outputs of the producer project.

[Parallel project execution](#44sinio)

With more and more CPU cores available on developer desktops and CI servers, it is important that Gradle is able to fully utilise these processing resources. More specifically, parallel execution attempts to:

* Reduce total build time for a multi-project build where execution is IO bound or otherwise does not consume all available CPU resources.
* Provide faster feedback for execution of small projects without awaiting completion of other projects.

Although Gradle already offers parallel test execution via [Test.setMaxParallelForks(int)](http://docs.google.com/javadoc/org/gradle/api/tasks/testing/Test.html#setMaxParallelForks-int-) the feature described in this section is parallel execution at a project level. Parallel execution is an incubating feature. Please use it and let us know how it works for you.

Parallel project execution allows the separate projects in a decoupled multi-project build to be executed in parallel (see also [Decoupled projects](#2jxsxqh)). While parallel execution does not strictly require decoupling at configuration time, the long-term goal is to provide a powerful set of features that will be available for fully decoupled projects. Such features include:

* [Configuration on-demand](#4i7ojhp).
* Configuration of projects in parallel.
* Re-use of configuration for unchanged projects.
* Project-level up-to-date checks.
* Using pre-built artifacts in the place of building dependent projects.

How does parallel execution work? First, you need to tell Gradle to use parallel mode. You can use the [--parallel command line argument](http://docs.google.com/command_line_interface.html#sec:command_line_performance) or configure your build environment ([Gradle properties](http://docs.google.com/build_environment.html#sec:gradle_configuration_properties)). Unless you provide a specific number of parallel threads, Gradle attempts to choose the right number based on available CPU cores. Every parallel worker exclusively owns a given project while executing a task. Task dependencies are fully supported and parallel workers will start executing upstream tasks first. Bear in mind that the alphabetical ordering of decoupled tasks, as can be seen during sequential execution, is not guaranteed in parallel mode. In other words, in parallel mode tasks will run as soon as their dependencies complete *and a task worker is available to run them*, which may be earlier than they would start during a sequential build. You should make sure that task dependencies and task inputs/outputs are declared correctly to avoid ordering issues.

[Decoupled Projects](#2jxsxqh)

Gradle allows any project to access any other project during both the configuration and execution phases. While this provides a great deal of power and flexibility to the build author, it also limits the flexibility that Gradle has when building those projects. For instance, this effectively prevents Gradle from correctly building multiple projects in parallel, configuring only a subset of projects, or from substituting a pre-built artifact in place of a project dependency.

Two projects are said to be *decoupled* if they do not directly access each other’s project model. Decoupled projects may only interact in terms of declared dependencies: [project dependencies](http://docs.google.com/dependency_types.html#sub:project_dependencies) and/or [task dependencies](http://docs.google.com/tutorial_using_tasks.html#sec:task_dependencies). Any other form of project interaction (i.e. by modifying another project object or by reading a value from another project object) causes the projects to be coupled. The consequence of coupling during the configuration phase is that if gradle is invoked with the 'configuration on demand' option, the result of the build can be flawed in several ways. The consequence of coupling during execution phase is that if gradle is invoked with the parallel option, one project task runs too late to influence a task of a project building in parallel. Gradle does not attempt to detect coupling and warn the user, as there are too many possibilities to introduce coupling.

A very common way for projects to be coupled is by using [configuration injection](#4d34og8). It may not be immediately apparent, but using key Gradle features like the allprojects and subprojects keywords automatically cause your projects to be coupled. This is because these keywords are used in a build.gradle file, which defines a project. Often this is a “root project” that does nothing more than define common configuration, but as far as Gradle is concerned this root project is still a fully-fledged project, and by using allprojects that project is effectively coupled to all other projects. Coupling of the root project to subprojects does not impact 'configuration on demand', but using the allprojects and subprojects in any subproject’s build.gradle file will have an impact.

This means that using any form of shared build script logic or configuration injection (allprojects, subprojects, etc.) will cause your projects to be coupled. As we extend the concept of project decoupling and provide features that take advantage of decoupled projects, we will also introduce new features to help you to solve common use cases (like configuration injection) without causing your projects to be coupled.

In order to make good use of cross project configuration without running into issues for parallel and 'configuration on demand' options, follow these recommendations:

* Avoid a subproject’s build.gradle referencing other subprojects; preferring cross configuration from the root project.
* Avoid changing the configuration of other projects at execution time.

[Multi-Project Building and Testing](#z337ya)

The build task of the Java plugin is typically used to compile, test, and perform code style checks (if the CodeQuality plugin is used) of a single project. In multi-project builds you may often want to do all of these tasks across a range of projects. The buildNeeded and buildDependents tasks can help with this.

In [this example](#1mrcu09), the “:services:personservice” project depends on both the “:api” and “:shared” projects. The “:api” project also depends on the “:shared” project.

Assume you are working on a single project, the “:api” project. You have been making changes, but have not built the entire project since performing a clean. You want to build any necessary supporting jars, but only perform code quality and unit tests on the project you have changed. The build task does this.

[Example: Build and Test Single Project](#206ipza)

**Output of** gradle :api:build

> gradle :api:build  
> Task :shared:compileJava  
> Task :shared:processResources  
> Task :shared:classes  
> Task :shared:jar  
> Task :api:compileJava  
> Task :api:processResources  
> Task :api:classes  
> Task :api:jar  
> Task :api:assemble  
> Task :api:compileTestJava  
> Task :api:processTestResources  
> Task :api:testClasses  
> Task :api:test  
> Task :api:check  
> Task :api:build  
  
BUILD SUCCESSFUL in 0s  
9 actionable tasks: 9 executed

If you have just gotten the latest version of source from your version control system which included changes in other projects that “:api” depends on, you might want to not only build all the projects you depend on, but test them as well. The buildNeeded task also tests all the projects from the project lib dependencies of the testRuntime configuration.

[Example: Build and Test Depended On Projects](#4k668n3)

**Output of** gradle :api:buildNeeded

> gradle :api:buildNeeded  
> Task :shared:compileJava  
> Task :shared:processResources  
> Task :shared:classes  
> Task :shared:jar  
> Task :api:compileJava  
> Task :api:processResources  
> Task :api:classes  
> Task :api:jar  
> Task :api:assemble  
> Task :api:compileTestJava  
> Task :api:processTestResources  
> Task :api:testClasses  
> Task :api:test  
> Task :api:check  
> Task :api:build  
> Task :shared:assemble  
> Task :shared:compileTestJava  
> Task :shared:processTestResources  
> Task :shared:testClasses  
> Task :shared:test  
> Task :shared:check  
> Task :shared:build  
> Task :shared:buildNeeded  
> Task :api:buildNeeded  
  
BUILD SUCCESSFUL in 0s  
12 actionable tasks: 12 executed

You also might want to refactor some part of the “:api” project that is used in other projects. If you make these types of changes, it is not sufficient to test just the “:api” project, you also need to test all projects that depend on the “:api” project. The buildDependents task also tests all the projects that have a project lib dependency (in the testRuntime configuration) on the specified project.

[Example: Build and Test Dependent Projects](#2zbgiuw)

**Output of** gradle :api:buildDependents

> gradle :api:buildDependents  
> Task :shared:compileJava  
> Task :shared:processResources  
> Task :shared:classes  
> Task :shared:jar  
> Task :api:compileJava  
> Task :api:processResources  
> Task :api:classes  
> Task :api:jar  
> Task :api:assemble  
> Task :api:compileTestJava  
> Task :api:processTestResources  
> Task :api:testClasses  
> Task :api:test  
> Task :api:check  
> Task :api:build  
> Task :services:personService:compileJava  
> Task :services:personService:processResources  
> Task :services:personService:classes  
> Task :services:personService:jar  
> Task :services:personService:assemble  
> Task :services:personService:compileTestJava  
> Task :services:personService:processTestResources  
> Task :services:personService:testClasses  
> Task :services:personService:test  
> Task :services:personService:check  
> Task :services:personService:build  
> Task :services:personService:buildDependents  
> Task :api:buildDependents  
  
BUILD SUCCESSFUL in 0s  
17 actionable tasks: 17 executed

Finally, you may want to build and test everything in all projects. Any task you run in the root project folder will cause that same named task to be run on all the children. So you can just run “gradle build” to build and test all projects.

[Multi Project and buildSrc](#3j2qqm3)

[Using buildSrc to organize build logic](http://docs.google.com/organizing_gradle_projects.html#sec:build_sources) tells us that we can place build logic to be compiled and tested in the special buildSrc directory. In a multi project build, there can only be one buildSrc directory which must be located in the root directory.

[1](#1egqt2p). The real use case we had, was using <http://lucene.apache.org/solr>, where you need a separate war for each index you are accessing. That was one reason why we have created a distribution of webapps. The Resin servlet container allows us, to let such a distribution point to a base installation of the servlet container.

[2](#3ygebqi). We do this here, as it makes the layout a bit easier. We usually put the project specific stuff into the build script of the respective projects.

Docs

* [User Manual](http://docs.google.com/userguide/userguide.html)
* [DSL Reference](http://docs.google.com/dsl/)
* [Release Notes](http://docs.google.com/release-notes.html)
* [Javadoc](http://docs.google.com/javadoc/)

News

* [Blog](https://blog.gradle.org/)
* [Newsletter](https://newsletter.gradle.com/)
* [Twitter](https://twitter.com/gradle)

Products

* [Build Scans](https://gradle.com/build-scans)
* [Build Cache](https://gradle.com/build-cache)
* [Enterprise Docs](https://gradle.com/enterprise/resources)

Get Help

* [Forums](https://discuss.gradle.org/c/help-discuss)
* [GitHub](https://github.com/gradle/)
* [Training](https://gradle.org/training/)
* [Services](https://gradle.org/services/)

Subscribe for important Gradle updates and news

Subscribe

By entering your email, you agree to our [Terms](https://gradle.org/terms/) and [Privacy Policy](https://gradle.org/privacy/), including receipt of emails. You can unsubscribe at any time.

© [Gradle Inc.](https://gradle.com) 2018 All rights reserved.

[Careers](https://gradle.com/careers) | [Privacy](https://gradle.org/privacy) | [Terms of Service](https://gradle.org/terms) | [Contact](https://gradle.org/contact/)