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/)

# Customizing Dependency Resolution Behavior

Contents

[Using dependency resolve rules](#4d34og8)

[Using dependency substitution rules](#2s8eyo1)

[Using component metadata rules](#17dp8vu)

[Using component selection rules](#3rdcrjn)

[Using module replacement rules](#26in1rg)

[Specifying default dependencies for a configuration](#lnxbz9)

[Enabling Ivy dynamic resolve mode](#35nkun2)

There are a number of ways that you can influence how Gradle resolves dependencies. All of these mechanisms offer an API to define a reason for why they are used. Providing reasons makes dependency resolution results more understandable. If any customization influenced the resolution result, the provided reason will show up in [dependency insight](http://docs.google.com/inspecting_dependencies.html#sec:identifying_reason_dependency_selection) report.

[Using dependency resolve rules](#4d34og8)

A dependency resolve rule is executed for each resolved dependency, and offers a powerful api for manipulating a requested dependency prior to that dependency being resolved. The feature currently offers the ability to change the group, name and/or version of a requested dependency, allowing a dependency to be substituted with a completely different module during resolution.

Dependency resolve rules provide a very powerful way to control the dependency resolution process, and can be used to implement all sorts of advanced patterns in dependency management. Some of these patterns are outlined below. For more information and code samples see the [ResolutionStrategy](http://docs.google.com/dsl/org.gradle.api.artifacts.ResolutionStrategy.html) class in the API documentation.

[Modelling releasable units](#1ksv4uv)

Often an organisation publishes a set of libraries with a single version; where the libraries are built, tested and published together. These libraries form a "releasable unit", designed and intended to be used as a whole. It does not make sense to use libraries from different releasable units together.

But it is easy for transitive dependency resolution to violate this contract. For example:

* module-a depends on releasable-unit:part-one:1.0
* module-b depends on releasable-unit:part-two:1.1

A build depending on both module-a and module-b will obtain different versions of libraries within the releasable unit.

Dependency resolve rules give you the power to enforce releasable units in your build. Imagine a releasable unit defined by all libraries that have org.gradle group. We can force all of these libraries to use a consistent version:

[Example: Forcing a consistent version for a group of libraries](#44sinio)

**build.gradle**

configurations.all {  
 resolutionStrategy.eachDependency { DependencyResolveDetails details ->  
 if (details.requested.group == 'org.gradle') {  
 details.useVersion '1.4'  
 details.because 'API breakage in higher versions'  
 }  
 }  
}

[Implementing a custom versioning scheme](#2jxsxqh)

In some corporate environments, the list of module versions that can be declared in Gradle builds is maintained and audited externally. Dependency resolve rules provide a neat implementation of this pattern:

* In the build script, the developer declares dependencies with the module group and name, but uses a placeholder version, for example: default.
* The default version is resolved to a specific version via a dependency resolve rule, which looks up the version in a corporate catalog of approved modules.

This rule implementation can be neatly encapsulated in a corporate plugin, and shared across all builds within the organisation.

[Example: Using a custom versioning scheme](#z337ya)

**build.gradle**

configurations.all {  
 resolutionStrategy.eachDependency { DependencyResolveDetails details ->  
 if (details.requested.version == 'default') {  
 def version = findDefaultVersionInCatalog(details.requested.group, details.requested.name)  
 details.useVersion version.version  
 details.because version.because  
 }  
 }  
}  
  
def findDefaultVersionInCatalog(String group, String name) {  
 //some custom logic that resolves the default version into a specific version  
 [version: "1.0", because: 'tested by QA']  
}

[Blacklisting a particular version with a replacement](#3j2qqm3)

Dependency resolve rules provide a mechanism for blacklisting a particular version of a dependency and providing a replacement version. This can be useful if a certain dependency version is broken and should not be used, where a dependency resolve rule causes this version to be replaced with a known good version. One example of a broken module is one that declares a dependency on a library that cannot be found in any of the public repositories, but there are many other reasons why a particular module version is unwanted and a different version is preferred.

In example below, imagine that version 1.2.1 contains important fixes and should always be used in preference to 1.2. The rule provided will enforce just this: any time version 1.2 is encountered it will be replaced with 1.2.1. Note that this is different from a forced version as described above, in that any other versions of this module would not be affected. This means that the 'newest' conflict resolution strategy would still select version 1.3 if this version was also pulled transitively.

[Example: Blacklisting a version with a replacement](#1y810tw)

**build.gradle**

configurations.all {  
 resolutionStrategy.eachDependency { DependencyResolveDetails details ->  
 if (details.requested.group == 'org.software' && details.requested.name == 'some-library' && details.requested.version == '1.2') {  
 details.useVersion '1.2.1'  
 details.because 'fixes critical bug in 1.2'  
 }  
 }  
}

[Substituting a dependency module with a compatible replacement](#4i7ojhp)

At times a completely different module can serve as a replacement for a requested module dependency. Examples include using groovy in place of groovy-all, or using log4j-over-slf4j instead of log4j. You can perform these substitutions using dependency resolve rules:

[Example: Changing dependency group and/or name during resolution](#2xcytpi)

**build.gradle**

configurations.all {  
 resolutionStrategy.eachDependency { DependencyResolveDetails details ->  
 if (details.requested.name == 'groovy-all') {  
 details.useTarget group: details.requested.group, name: 'groovy', version: details.requested.version  
 details.because "prefer 'groovy' over 'groovy-all'"  
 }  
 if (details.requested.name == 'log4j') {  
 details.useTarget "org.slf4j:log4j-over-slf4j:1.7.10"  
 details.because "prefer 'log4j-over-slf4j' 1.7.10 over any version of 'log4j'"  
 }  
 }  
}

[Using dependency substitution rules](#2s8eyo1)

Dependency substitution rules work similarly to dependency resolve rules. In fact, many capabilities of dependency resolve rules can be implemented with dependency substitution rules. They allow project and module dependencies to be transparently substituted with specified replacements. Unlike dependency resolve rules, dependency substitution rules allow project and module dependencies to be substituted interchangeably.

*Adding a dependency substitution rule to a configuration changes the timing of when that configuration is resolved.* Instead of being resolved on first use, the configuration is instead resolved when the task graph is being constructed. This can have unexpected consequences if the configuration is being further modified during task execution, or if the configuration relies on modules that are published during execution of another task.

To explain:

* A Configuration can be declared as an input to any Task, and that configuration can include project dependencies when it is resolved.
* If a project dependency is an input to a Task (via a configuration), then tasks to build the project artifacts must be added to the task dependencies.
* In order to determine the project dependencies that are inputs to a task, Gradle needs to resolve the Configuration inputs.
* Because the Gradle task graph is fixed once task execution has commenced, Gradle needs to perform this resolution prior to executing any tasks.

In the absence of dependency substitution rules, Gradle knows that an external module dependency will never transitively reference a project dependency. This makes it easy to determine the full set of project dependencies for a configuration through simple graph traversal. With this functionality, Gradle can no longer make this assumption, and must perform a full resolve in order to determine the project dependencies.

[Substituting an external module dependency with a project dependency](#1ci93xb)

One use case for dependency substitution is to use a locally developed version of a module in place of one that is downloaded from an external repository. This could be useful for testing a local, patched version of a dependency.

The module to be replaced can be declared with or without a version specified.

[Example: Substituting a module with a project](#3whwml4)

**build.gradle**

configurations.all {  
 resolutionStrategy.dependencySubstitution {  
 substitute module("org.utils:api") with project(":api") because "we work with the unreleased development version"  
 substitute module("org.utils:util:2.5") with project(":util")  
 }  
}

Note that a project that is substituted must be included in the multi-project build (via settings.gradle). Dependency substitution rules take care of replacing the module dependency with the project dependency and wiring up any task dependencies, but do not implicitly include the project in the build.

[Substituting a project dependency with a module replacement](#2bn6wsx)

Another way to use substitution rules is to replace a project dependency with a module in a multi-project build. This can be useful to speed up development with a large multi-project build, by allowing a subset of the project dependencies to be downloaded from a repository rather than being built.

The module to be used as a replacement must be declared with a version specified.

[Example: Substituting a project with a module](#qsh70q)

**build.gradle**

configurations.all {  
 resolutionStrategy.dependencySubstitution {  
 substitute project(":api") with module("org.utils:api:1.3") because "we use a stable version of utils"  
 }  
}

When a project dependency has been replaced with a module dependency, that project is still included in the overall multi-project build. However, tasks to build the replaced dependency will not be executed in order to build the resolve the depending Configuration.

[Conditionally substituting a dependency](#3as4poj)

A common use case for dependency substitution is to allow more flexible assembly of sub-projects within a multi-project build. This can be useful for developing a local, patched version of an external dependency or for building a subset of the modules within a large multi-project build.

The following example uses a dependency substitution rule to replace any module dependency with the group org.example, but only if a local project matching the dependency name can be located.

[Example: Conditionally substituting a dependency](#1pxezwc)

**build.gradle**

configurations.all {  
 resolutionStrategy.dependencySubstitution.all { DependencySubstitution dependency ->  
 if (dependency.requested instanceof ModuleComponentSelector && dependency.requested.group == "org.example") {  
 def targetProject = findProject(":${dependency.requested.module}")  
 if (targetProject != null) {  
 dependency.useTarget targetProject  
 }  
 }  
 }  
 }

Note that a project that is substituted must be included in the multi-project build (via settings.gradle). Dependency substitution rules take care of replacing the module dependency with the project dependency, but do not implicitly include the project in the build

[Using component metadata rules](#17dp8vu)

Each module has metadata associated with it, such as its group, name, version, dependencies, and so on. This metadata typically originates in the module’s descriptor. Metadata rules allow certain parts of a module’s metadata to be manipulated from within the build script. They take effect after a module’s descriptor has been downloaded, but before it has been selected among all candidate versions. This makes metadata rules another instrument for customizing dependency resolution.

One piece of module metadata that Gradle understands is a module’s *status scheme*. This concept, also known from Ivy, models the different levels of maturity that a module transitions through over time. The default status scheme, ordered from least to most mature status, is integration, milestone, release. Apart from a status scheme, a module also has a (current) *status*, which must be one of the values in its status scheme. If not specified in the (Ivy) descriptor, the status defaults to integration for Ivy modules and Maven snapshot modules, and release for Maven modules that aren’t snapshots.

A module’s status and status scheme are taken into consideration when a latest version selector is resolved. Specifically, latest.someStatus will resolve to the highest module version that has status someStatus or a more mature status. For example, with the default status scheme in place, latest.integration will select the highest module version regardless of its status (because integration is the least mature status), whereas latest.release will select the highest module version with status release. Here is what this looks like in code:

[Example: 'Latest' version selector](#49x2ik5)

**build.gradle**

dependencies {  
 config1 "org.sample:client:latest.integration"  
 config2 "org.sample:client:latest.release"  
}  
  
task listConfigs {  
 doLast {  
 configurations.config1.each { println it.name }  
 println()  
 configurations.config2.each { println it.name }  
 }  
}

**Output of** gradle -q listConfigs

> gradle -q listConfigs  
client-1.5.jar  
  
client-1.4.jar

The next example demonstrates latest selectors based on a custom status scheme declared in a component metadata rule that applies to all modules:

[Example: Custom status scheme](#2p2csry)

**build.gradle**

class CustomStatusRule implements ComponentMetadataRule {  
 @Override  
 void execute(ComponentMetadataContext context) {  
 def details = context.details  
 if (details.id.group == "org.sample" && details.id.name == "api") {  
 details.statusScheme = ["bronze", "silver", "gold", "platinum"]  
 }  
 }  
}  
  
dependencies {  
 config3 "org.sample:api:latest.silver"  
 components {  
 all(CustomStatusRule)  
 }  
}

Component metadata rules can be applied to a specified module. Modules must be specified in the form of group:module.

[Example: Custom status scheme by module](#147n2zr)

**build.gradle**

class ModuleStatusRule implements ComponentMetadataRule {  
 @Override  
 void execute(ComponentMetadataContext context) {  
 context.details.statusScheme = ["int", "rc", "prod"]  
 }  
}  
  
dependencies {  
 config4 "org.sample:lib:latest.prod"  
 components {  
 withModule('org.sample:lib', ModuleStatusRule)  
 }  
}

Gradle can also provide to component metadata rules the Ivy-specific metadata for modules resolved from an Ivy repository. Values from the Ivy descriptor are made available via the [IvyModuleDescriptor](http://docs.google.com/javadoc/org/gradle/api/artifacts/ivy/IvyModuleDescriptor.html) interface.

[Example: Ivy component metadata rule](#3o7alnk)

**build.gradle**

class IvyComponentRule implements ComponentMetadataRule {  
 @Override  
 void execute(ComponentMetadataContext context) {  
 def descriptor = context.getDescriptor(IvyModuleDescriptor)  
 if (descriptor != null && descriptor.branch == 'testing') {  
 context.details.status = "rc"  
 }  
 }  
}  
dependencies {  
 config5 "org.sample:lib:latest.rc"  
 components {  
 withModule("org.sample:lib", IvyComponentRule)  
 }  
}

Note that while any rule can request the [IvyModuleDescriptor](http://docs.google.com/javadoc/org/gradle/api/artifacts/ivy/IvyModuleDescriptor.html), only components sourced from an Ivy repository will have a non-null value for it.

As can be seen in the examples above, component metadata rules are defined by implementing [ComponentMetadataRule](http://docs.google.com/javadoc/org/gradle/api/artifacts/ComponentMetadataRule.html) which has a single execute method receiving an instance of [ComponentMetadataContext](http://docs.google.com/javadoc/org/gradle/api/artifacts/ComponentMetadataContext.html) as parameter.

The next example shows how you can configure the ComponentMetadataRule through an [ActionConfiguration](http://docs.google.com/javadoc/org/gradle/api/ActionConfiguration.html).

[Example: Configuration of ComponentMetadataRule](#23ckvvd)

**build.gradle**

class ConfiguredRule implements ComponentMetadataRule {  
 String param  
  
 @javax.inject.Inject  
 ConfiguredRule(String param) {  
 this.param = param  
 }  
  
 @Override  
 void execute(ComponentMetadataContext context) {  
 if (param == 'sampleValue') {  
 context.details.statusScheme = ["bronze", "silver", "gold", "platinum"]  
 }  
 }  
}  
dependencies {  
 config6 "org.sample:api:latest.gold"  
 components {  
 withModule('org.sample:api', ConfiguredRule, {  
 params('sampleValue')  
 })  
 }  
}

This happens by having a constructor in your implementation of ComponentMetadataRule accepting the parameters that were configured and the services that need injecting.

Gradle enforces isolation of instances of ComponentMetadataRule. This means that all passed in parameters must be Serializable or known Gradle types that can be isolated.

In addition, Gradle services can be injected into your ComponentMetadataRule. This is for the moment limited to the [RepositoryResourceAccessor](http://docs.google.com/javadoc/org/gradle/api/artifacts/repositories/RepositoryResourceAccessor.html). Because of this, the moment you have a constructor, it must be annotated with @javax.inject.Inject.

[Using component selection rules](#3rdcrjn)

Component selection rules may influence which component instance should be selected when multiple versions are available that match a version selector. Rules are applied against every available version and allow the version to be explicitly rejected by rule. This allows Gradle to ignore any component instance that does not satisfy conditions set by the rule. Examples include:

* For a dynamic version like 1.+ certain versions may be explicitly rejected from selection.
* For a static version like 1.4 an instance may be rejected based on extra component metadata such as the Ivy branch attribute, allowing an instance from a subsequent repository to be used.

Rules are configured via the [ComponentSelectionRules](http://docs.google.com/dsl/org.gradle.api.artifacts.ComponentSelectionRules.html) object. Each rule configured will be called with a [ComponentSelection](http://docs.google.com/dsl/org.gradle.api.artifacts.ComponentSelection.html) object as an argument which contains information about the candidate version being considered. Calling [ComponentSelection.reject(java.lang.String)](http://docs.google.com/dsl/org.gradle.api.artifacts.ComponentSelection.html#org.gradle.api.artifacts.ComponentSelection:reject(java.lang.String)) causes the given candidate version to be explicitly rejected, in which case the candidate will not be considered for the selector.

The following example shows a rule that disallows a particular version of a module but allows the dynamic version to choose the next best candidate.

[Example: Component selection rule](#ihv636)

**build.gradle**

configurations {  
 rejectConfig {  
 resolutionStrategy {  
 componentSelection {  
 // Accept the highest version matching the requested version that isn't '1.5'  
 all { ComponentSelection selection ->  
 if (selection.candidate.group == 'org.sample' && selection.candidate.module == 'api' && selection.candidate.version == '1.5') {  
 selection.reject("version 1.5 is broken for 'org.sample:api'")  
 }  
 }  
 }  
 }  
 }  
}  
  
dependencies {  
 rejectConfig "org.sample:api:1.+"  
}

Note that version selection is applied starting with the highest version first. The version selected will be the first version found that all component selection rules accept. A version is considered accepted if no rule explicitly rejects it.

Similarly, rules can be targeted at specific modules. Modules must be specified in the form of group:module.

[Example: Component selection rule with module target](#32hioqz)

**build.gradle**

configurations {  
 targetConfig {  
 resolutionStrategy {  
 componentSelection {  
 withModule("org.sample:api") { ComponentSelection selection ->  
 if (selection.candidate.version == "1.5") {  
 selection.reject("version 1.5 is broken for 'org.sample:api'")  
 }  
 }  
 }  
 }  
 }  
}

Component selection rules can also consider component metadata when selecting a version. Possible metadata arguments that can be considered are [ComponentMetadata](http://docs.google.com/javadoc/org/gradle/api/artifacts/ComponentMetadata.html) and [IvyModuleDescriptor](http://docs.google.com/javadoc/org/gradle/api/artifacts/ivy/IvyModuleDescriptor.html).

[Example: Component selection rule with metadata](#1hmsyys)

**build.gradle**

configurations {  
 metadataRulesConfig {  
 resolutionStrategy {  
 componentSelection {  
 // Reject any versions with a status of 'experimental'  
 all { ComponentSelection selection, ComponentMetadata metadata ->  
 if (selection.candidate.group == 'org.sample' && metadata.status == 'experimental') {  
 selection.reject("don't use experimental candidates from 'org.sample'")  
 }  
 }  
 // Accept the highest version with either a "release" branch or a status of 'milestone'  
 withModule('org.sample:api') { ComponentSelection selection, IvyModuleDescriptor descriptor, ComponentMetadata metadata ->  
 if (descriptor.branch != "release" && metadata.status != 'milestone') {  
 selection.reject("'org.sample:api' must have testing branch or milestone status")  
 }  
 }  
 }  
 }  
 }  
}

Note that a [ComponentSelection](http://docs.google.com/dsl/org.gradle.api.artifacts.ComponentSelection.html) argument is *always* required as the first parameter when declaring a component selection rule with additional Ivy metadata parameters, but the metadata parameters can be declared in any order.

Lastly, component selection rules can also be defined using a *rule source* object. A rule source object is any object that contains exactly one method that defines the rule action and is annotated with @Mutate.

This method:

* must return void.
* must have [ComponentSelection](http://docs.google.com/dsl/org.gradle.api.artifacts.ComponentSelection.html) as the first argument.
* may have additional parameters of type [ComponentMetadata](http://docs.google.com/javadoc/org/gradle/api/artifacts/ComponentMetadata.html) and/or [IvyModuleDescriptor](http://docs.google.com/javadoc/org/gradle/api/artifacts/ivy/IvyModuleDescriptor.html).

[Example: Component selection rule using a rule source object](#41mghml)

**build.gradle**

class RejectTestBranch {  
 @Mutate  
 void evaluateRule(ComponentSelection selection, IvyModuleDescriptor ivy) {  
 if (ivy.branch == "test") {  
 selection.reject("reject test branch")  
 }  
 }  
}  
  
configurations {  
 ruleSourceConfig {  
 resolutionStrategy {  
 componentSelection {  
 all new RejectTestBranch()  
 }  
 }  
 }  
}

[Using module replacement rules](#26in1rg)

Module replacement rules allow a build to declare that a legacy library has been replaced by a new one. A good example when a new library replaced a legacy one is the google-collections -> guava migration. The team that created google-collections decided to change the module name from com.google.collections:google-collections into com.google.guava:guava. This is a legal scenario in the industry: teams need to be able to change the names of products they maintain, including the module coordinates. Renaming of the module coordinates has impact on conflict resolution.

To explain the impact on conflict resolution, let’s consider the google-collections -> guava scenario. It may happen that both libraries are pulled into the same dependency graph. For example, *our project* depends on guava but some of *our dependencies* pull in a legacy version of google-collections. This can cause runtime errors, for example during test or application execution. Gradle does not automatically resolve the google-collections -> guava conflict because it is not considered as a *version conflict*. It’s because the module coordinates for both libraries are completely different and conflict resolution is activated when group and module coordinates are the same but there are different versions available in the dependency graph (for more info, refer to the section on conflict resolution). Traditional remedies to this problem are:

* Declare exclusion rule to avoid pulling in google-collections to graph. It is probably the most popular approach.
* Avoid dependencies that pull in legacy libraries.
* Upgrade the dependency version if the new version no longer pulls in a legacy library.
* Downgrade to google-collections. It’s not recommended, just mentioned for completeness.

Traditional approaches work but they are not general enough. For example, an organisation wants to resolve the google-collections -> guava conflict resolution problem in all projects. Starting from Gradle 2.2 it is possible to declare that certain module was replaced by other. This enables organisations to include the information about module replacement in the corporate plugin suite and resolve the problem holistically for all Gradle-powered projects in the enterprise.

[Example: Declaring a module replacement](#2grqrue)

**build.gradle**

dependencies {  
 modules {  
 module("com.google.collections:google-collections") {  
 replacedBy("com.google.guava:guava", "google-collections is now part of Guava")  
 }  
 }  
}

For more examples and detailed API, refer to the DSL reference for [ComponentMetadataHandler](http://docs.google.com/javadoc/org/gradle/api/artifacts/dsl/ComponentMetadataHandler.html).

What happens when we declare that google-collections is replaced by guava? Gradle can use this information for conflict resolution. Gradle will consider every version of guava newer/better than any version of google-collections. Also, Gradle will ensure that only guava jar is present in the classpath / resolved file list. Note that if only google-collections appears in the dependency graph (e.g. no guava) Gradle will not eagerly replace it with guava. Module replacement is an information that Gradle uses for resolving conflicts. If there is no conflict (e.g. only google-collections or only guava in the graph) the replacement information is not used.

Currently it is not possible to declare that a given module is replaced by a set of modules. However, it is possible to declare that multiple modules are replaced by a single module.

[Specifying default dependencies for a configuration](#lnxbz9)

A configuration can be configured with default dependencies to be used if no dependencies are explicitly set for the configuration. A primary use case of this functionality is for developing plugins that make use of versioned tools that the user might override. By specifying default dependencies, the plugin can use a default version of the tool only if the user has not specified a particular version to use.

[Example: Specifying default dependencies on a configuration](#vx1227)

**build.gradle**

configurations {  
 pluginTool {  
 defaultDependencies { dependencies ->  
 dependencies.add(project.dependencies.create("org.gradle:my-util:1.0"))  
 }  
 }  
}

[Enabling Ivy dynamic resolve mode](#35nkun2)

Gradle’s Ivy repository implementations support the equivalent to Ivy’s dynamic resolve mode. Normally, Gradle will use the rev attribute for each dependency definition included in an ivy.xml file. In dynamic resolve mode, Gradle will instead prefer the revConstraint attribute over the rev attribute for a given dependency definition. If the revConstraint attribute is not present, the rev attribute is used instead.

To enable dynamic resolve mode, you need to set the appropriate option on the repository definition. A couple of examples are shown below. Note that dynamic resolve mode is only available for Gradle’s Ivy repositories. It is not available for Maven repositories, or custom Ivy DependencyResolver implementations.

[Example: Enabling dynamic resolve mode](#3fwokq0)

**build.gradle**

// Can enable dynamic resolve mode when you define the repository  
repositories {  
 ivy {  
 url "http://repo.mycompany.com/repo"  
 resolve.dynamicMode = true  
 }  
}  
  
// Can use a rule instead to enable (or disable) dynamic resolve mode for all repositories  
repositories.withType(IvyArtifactRepository) {  
 resolve.dynamicMode = true  
}

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/)