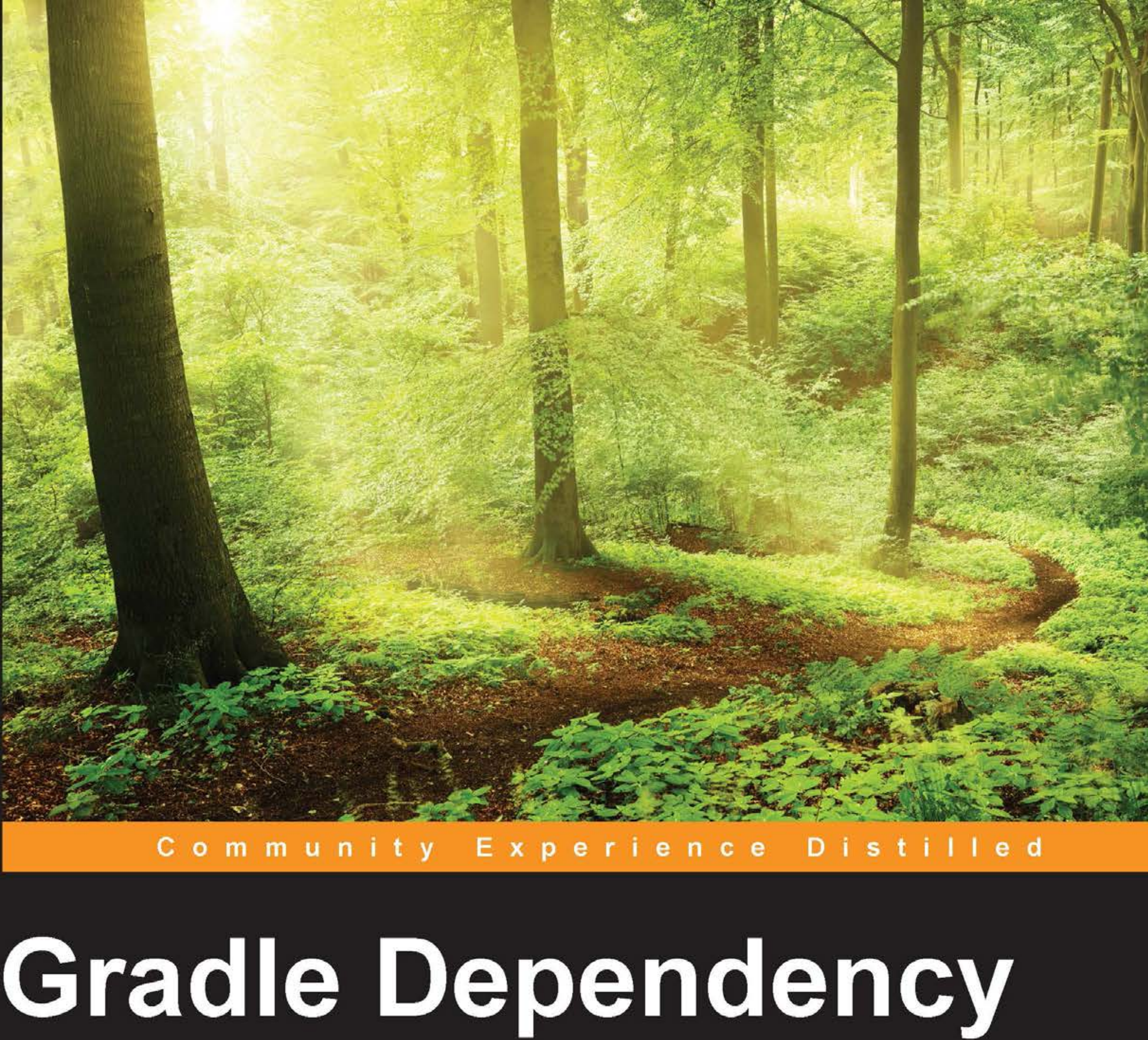

Community Experience Distilled

Gradle Dependency Management

Learn how to use Gradle's powerful dependency management through extensive code samples, and discover how to define, customize, and deploy dependencies

Hubert Klein Ikkink

[PACKT] open source*
PUBLISHING
community experience distilled

Gradle Dependency Management

Learn how to use Gradle's powerful dependency management through extensive code samples, and discover how to define, customize, and deploy dependencies

Hubert Klein Ikkink

[PACKT] PUBLISHING open source
community experience distilled

BIRMINGHAM - MUMBAI

Gradle Dependency Management

First published: June 2015

Production reference: 1120615

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78439-278-9

www.packtpub.com

Credits

Author
Hubert Klein Ikkink

Reviewers
Tony Dieppa
Izzet Mustafaiev
Konstantin Zgirovskiy

Commissioning Editor
Pramila Balan

Acquisition Editor
Sonali Vernekar

Content Development Editor
Athira Laji

Technical Editor
Siddhesh Ghadi

Copy Editor
Sarang Chari

Project Coordinator
Harshal Ved

Proofreader
Safis Editing

Indexer
Monica Mehta

Production Coordinator
Arvindkumar Gupta

Cover Work
Arvindkumar Gupta

About the Author

Hubert Klein Ikkink, born in 1973, lives in Tilburg, the Netherlands, with his beautiful wife and three gorgeous children. He is also known as mrhaki, which is simply the initials of his name prepended by "mr". He studied information systems and management at Tilburg University. After finishing his studies in 1996, he started to develop Java software. Over the years, his focus switched from applets to servlets, and from Java Enterprise Edition applications to Spring-based software and Groovy-related technologies. He likes the expressiveness of the Groovy language and how it is used in other tools, such as Gradle. He also wrote *Gradle Effective Implementation Guide*, *Packt Publishing*.

In the Netherlands, Hubert works for a company called JDriven. JDriven focuses on technologies that simplify and improve the development of enterprise applications. Employees of JDriven have years of experience with Java and related technologies and are all eager to learn about new technologies. Hubert works on projects using Grails and Java combined with Groovy and Gradle.

About the Reviewers

Izzet Mustafaiev is a family guy who likes to throw BBQ parties and travel.

Professionally, he is a software engineer working at EPAM Systems with primary skills in Java and hands-on experience in Groovy/Ruby, and is exploring FP with Erlang/Elixir. Izzet has participated in different projects as a developer and as an architect. He advocates XP, clean code, and DevOps practices when he speaks at engineering conferences.

Konstantin Zgirovskiy grew up alongside Android, in a manner of speaking. In 2008, he started programming web services and Chrome extensions for an online browser game, which was later made official. In 2011, Konstantin continued to explore Android through writing a game, which brought him victory in a local programming contest. Nowadays, he works at Looksery, Inc., where he is involved in developing an app with face-tracking and transformation technology for video chats, video selfies, and images on mobile devices.

I would like to thank my cats, friends, colleagues, and family for their support. My thanks also go to Packt Publishing for this opportunity. Additionally, I would like to thank Dasha Tsareva-Lenskaya for motivating and encouraging me whenever I got distracted.

Table of Contents

Preface

When we write code in our Java or Groovy project, we mostly have dependencies on other projects or libraries. For example, we could use the Spring framework in our project, so we are dependent on classes found in the Spring framework. We want to be able to manage such dependencies from Gradle, our build automation tool.

We will see how we can define and customize the dependencies we need. We learn not only how to define the dependencies, but also how to work with repositories that store the dependencies. Next, we will see how to customize the way Gradle resolves dependencies.

Besides being dependent on other libraries, our project can also be a dependency for other projects. This means that we need to know how to deploy our project artifacts so that other developers can use it. We learn how to define artifacts and how to deploy them to, for example, a Maven or Ivy repository.

What this book covers

Chapter 1, *Defining Dependencies,* introduces dependency configurations as a way to organize dependencies. You will learn about the different types of dependencies in Gradle.

Chapter 2, *Working with Repositories*, covers how we can define repositories that store our dependencies. We will see not only how to set the location, but also the layout of a repository.

Chapter 3, *Resolving Dependencies,* is about how Gradle resolves our dependencies. You will learn how to customize the dependency resolution and resolve conflicts between dependencies.

Chapter 4, *Publishing Artifacts*, covers how to define artifacts for our project to be published as dependencies for others. We will see how to use configurations to define artifacts. We also use a local directory as a repository to publish the artifacts.

Chapter 5, *Publishing to a Maven Repository*, looks at how to publish our artifacts to a Maven repository. You will learn how to define a publication for a Maven-like repository, such as Artifactory or Nexus, and how to use the new and incubating publishing feature of Gradle.

Chapter 6, *Publishing to Bintray*, covers how to deploy our artifacts to Bintray. Bintray calls itself a Distribution as a Service and provides a low-level way to publish our artifacts to the world. In this chapter, we will look at how to use the Bintray Gradle plugin to publish our artifacts.

Chapter 7, *Publishing to an Ivy Repository*, is about publishing our artifacts to an Ivy repository. We will look into the different options to publish our artifacts to an Ivy repository, which is actually quite similar to publishing to a Maven repository.

What you need for this book

In order to work with Gradle and the code samples in this book, we need at least Java Development Kit (version 1.6 or higher), Gradle (samples are written with Gradle 2.3), and a good text editor.

Who this book is for

This book is for you if you are working on Java or Groovy projects and are using, or are going to use, Gradle to build your code. If your code depends on other projects or libraries, you will learn how to define and customize those dependencies. Your code can also be used by other projects, so you want to publish your project as a dependency for others whom you want to read this book.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
// Define new configurations for build.
configurations {

    // Define configuration vehicles.
    vehicles {
        description = 'Contains vehicle dependencies'
    }

    traffic {
        extendsFrom vehicles
        description = 'Contains traffic dependencies'
    }

}
```

Any command-line input or output is written as follows:

```
$ gradle bintrayUpload
:generatePomFileForSamplePublication
:compileJava
:processResources UP-TO-DATE
:classes
:jar
:publishSamplePublicationToMavenLocal
:bintrayUpload

BUILD SUCCESSFUL

Total time: 9.125 secs
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "From this screen, we click on the **New package** button."

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/B03462_Coloredimages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Defining Dependencies

When we develop software, we need to write code. Our code consists of packages with classes, and those can be dependent on the other classes and packages in our project. This is fine for one project, but we sometimes depend on classes in other projects we didn't develop ourselves, for example, we might want to use classes from an Apache Commons library or we might be working on a project that is part of a bigger, multi-project application and we are dependent on classes in these other projects.

Most of the time, when we write software, we want to use classes outside of our project. Actually, we have a dependency on those classes. Those dependent classes are mostly stored in archive files, such as **Java Archive** (**JAR**) files. Such archive files are identified by a unique version number, so we can have a dependency on the library with a specific version.

In this chapter, you are going to learn how to define dependencies in your Gradle project. We will see how we can define the configurations of dependencies. You will learn about the different dependency types in Gradle and how to use them when you configure your build.

Declaring dependency configurations

In Gradle, we define dependency configurations to group dependencies together. A dependency configuration has a name and several properties, such as a description and is actually a special type of `FileCollection`. Configurations can extend from each other, so we can build a hierarchy of configurations in our build files. Gradle plugins can also add new configurations to our project, for example, the Java plugin adds several new configurations, such as `compile` and `testRuntime`, to our project. The `compile` configuration is then used to define the dependencies that are needed to compile our source tree. The dependency configurations are defined with a `configurations` configuration block. Inside the block, we can define new configurations for our build. All configurations are added to the project's `ConfigurationContainer` object.

In the following example build file, we define two new configurations, where the `traffic` configuration extends from the `vehicles` configuration. This means that any dependency added to the `vehicles` configuration is also available in the `traffic` configuration. We can also assign a `description` property to our configuration to provide some more information about the configuration for documentation purposes. The following code shows this:

```
// Define new configurations for build.
configurations {

  // Define configuration vehicles.
  vehicles {
    description = 'Contains vehicle dependencies'
  }

  traffic {
    extendsFrom vehicles
    description = 'Contains traffic dependencies'
  }

}
```

To see which configurations are available in a project, we can execute the `dependencies` task. This task is available for each Gradle project. The task outputs all the configurations and dependencies of a project. Let's run this task for our current project and check the output:

```
$ gradle -q dependencies

------------------------------------------------------------
Root project
------------------------------------------------------------

traffic - Contains traffic dependencies
No dependencies

vehicles - Contains vehicle dependencies
No dependencies
```

Note that we can see our two configurations, `traffic` and `vehicles`, in the output. We have not defined any dependencies to these configurations, as shown in the output.

The Java plugin adds a couple of configurations to a project, which are used by the tasks from the Java plugin. Let's add the Java plugin to our Gradle build file:

```
    apply plugin: 'java'
```

To see which configurations are added, we invoke the `dependencies` task and look at the output:

```
$ gradle -q dependencies

------------------------------------------------------------
Root project
------------------------------------------------------------

archives - Configuration for archive artifacts.
No dependencies

compile - Compile classpath for source set 'main'.
No dependencies

default - Configuration for default artifacts.
No dependencies

runtime - Runtime classpath for source set 'main'.
No dependencies

testCompile - Compile classpath for source set 'test'.
No dependencies

testRuntime - Runtime classpath for source set 'test'.
No dependencies
```

We see six configurations in our project just by adding the Java plugin. The `archives` configuration is used to group the artifacts our project creates. The other configurations are used to group the dependencies for our project. In the following table, the dependency configurations are summarized:

Name	**Extends**	**Description**
compile	none	These are dependencies to compile.
runtime	compile	These are runtime dependencies.
testCompile	compile	These are extra dependencies to compile tests.

Name	**Extends**	**Description**
testRuntime	runtime, testCompile	These are extra dependencies to run tests.
default	runtime	These are dependencies used by this project and artifacts created by this project.

Later in the chapter, we will see how we can work with the dependencies assigned to the configurations. In the next section, we will learn how to declare our project's dependencies.

Declaring dependencies

We defined configurations or applied a plugin that added new configurations to our project. However, a configuration is empty unless we add dependencies to the configuration. To declare dependencies in our Gradle build file, we must add the `dependencies` configuration block. The configuration block will contain the definition of our dependencies. In the following example Gradle build file, we define the `dependencies` block:

```
// Dependencies configuration block.
dependencies {
    // Here we define our dependencies.
}
```

Inside the configuration block, we use the name of a dependency configuration followed by the description of our dependencies. The name of the dependency configuration can be defined explicitly in the build file or can be added by a plugin we use. In Gradle, we can define several types of dependencies. In the following table, we will see the different types we can use:

Dependency type	**Description**
External module dependency	This is a dependency on an external module or library that is probably stored in a repository.
Client module dependency	This is a dependency on an external module where the artifacts are stored in a repository, but the meta information about the module is in the build file. We can override meta information using this type of dependency.
Project dependency	This is a dependency on another Gradle project in the same build.
File dependency	This is a dependency on a collection of files on the local computer.

Dependency type	Description
Gradle API dependency	This is a dependency on the Gradle API of the current Gradle version. We use this dependency when we develop Gradle plugins and tasks.
Local Groovy dependency	This is a dependency on the Groovy libraries used by the current Gradle version. We use this dependency when we develop Gradle plugins and tasks.

External module dependencies

External module dependencies are the most common dependencies in projects. These dependencies refer to a module in an external repository. Later in the book, we will find out more about repositories, but basically, a repository stores modules in a central location. A module contains one or more artifacts and meta information, such as references to the other modules it depends on.

We can use two notations to define an external module dependency in Gradle. We can use a string notation or a map notation. With the map notation, we can use all the properties available for a dependency. The string notation allows us to set a subset of the properties but with a very concise syntax.

In the following example Gradle build file, we define several dependencies using the string notation:

```
// Define dependencies.
dependencies {
  // Defining two dependencies.
  vehicles 'com.vehicles:car:1.0', 'com.vehicles:truck:2.0'

  // Single dependency.
  traffic 'com.traffic:pedestrian:1.0'
}
```

The string notation has the following format: **moduleGroup:moduleName:version**. Before the first colon, the module group name is used, followed by the module name, and the version is mentioned last.

If we use the map notation, we use the names of the attributes explicitly and set the value for each attribute. Let's rewrite our previous example build file and use the map notation:

```
// Compact definition of configurations.
configurations {
  vehicles
  traffic.extendsFrom vehicles
}

// Define dependencies.
dependencies {
  // Defining two dependencies.
  vehicles(
    [group: 'com.vehicles', name: 'car', version: '1.0'],
    [group: 'com.vehicles', name: 'truck', version: '2.0'],
  )

  // Single dependency.
  traffic group: 'com.traffic', name: 'pedestrian', version:
  '1.0'
}
```

We can specify extra configuration attributes with the map notation, or we can add an extra configuration closure. One of the attributes of an external module dependency is the `transitive` attribute. We learn more about how to work with transitive dependencies in *Chapter 3, Resolving Dependencies*. In the next example build file, we will set this attribute using the map notation and a configuration closure:

```
dependencies {
  // Use transitive attribute in map notation.
  vehicles group: 'com.vehicles', name: 'car',
      version: '1.0', transitive: false

  // Combine map notation with configuration closure.
  vehicles(group: 'com.vehicles', name: 'car', version: '1.0') {
    transitive = true
  }

  // Combine string notation with configuration closure.
  traffic('com.traffic:pedestrian:1.0') {
    transitive = false
  }
}
```

In the rest of this section, you will learn about more attributes you can use to configure a dependency.

Once of the advantages of Gradle is that we can write Groovy code in our build file. This means that we can define methods and variables and use them in other parts of our Gradle file. This way, we can even apply refactoring to our build file and make maintainable build scripts. Note that in our examples, we included multiple dependencies with the `com.vehicles` group name. The value is defined twice, but we can also create a new variable with the group name and reference of the variable in the dependencies configuration. We define a variable in our build file inside an `ext` configuration block. We use the `ext` block in Gradle to add extra properties to an object, such as our project.

The following sample code defines an extra variable to hold the group name:

```
// Define project property with
// dependency group name 'com.vehicles'
ext {
  groupNameVehicles = 'com.vehicles'
}

dependencies {
  // Using Groovy string support with
  // variable substition.
  vehicles "$groupNameVehicles:car:1.0"

  // Using map notation and reference
  // property groupNameVehicles.
  vehicles group: groupNameVehicles, name: 'truck', version:
  '2.0'
}
```

If we define an external module dependency, then Gradle tries to find a module descriptor in a repository. If the module descriptor is available, it is parsed to see which artifacts need to be downloaded. Also, if the module descriptor contains information about the dependencies needed by the module, those dependencies are downloaded as well. Sometimes, a dependency has no descriptor in the repository, and it is only then that Gradle downloads the artifact for that dependency.

A dependency based on a Maven module only contains one artifact, so it is easy for Gradle to know which artifact to download. But for a Gradle or Ivy module, it is not so obvious, because a module can contain multiple artifacts. The module will have multiple configurations, each with different artifacts. Gradle will use the configuration with the name `default` for such modules. So, any artifacts and dependencies associated with the `default` configuration are downloaded. However, it is possible that the `default` configuration doesn't contain the artifacts we need. We, therefore, can specify the `configuration` attribute for the dependency configuration to specify a specific configuration that we need.

The following example defines a `configuration` attribute for the dependency configuration:

```
dependencies {
  // Use the 'jar' configuration defined in the
  // module descriptor for this dependency.
  traffic group: 'com.traffic',
      name: 'pedestrian',
      version: '1.0',
      configuration: 'jar'

}
```

When there is no module descriptor for a dependency, only the artifact is downloaded by Gradle. We can use an artifact-only notation if we only want to download the artifact for a module with a descriptor and not any dependencies. Or, if we want to download another archive file, such as a TAR file, with documentation, from a repository.

To use the artifact-only notation, we must add the file extension to the dependency definition. If we use the string notation, we must add the extension prefixed with an `@` sign after the version. With the map notation, we can use the `ext` attribute to set the extension. If we define our dependency as artifact-only, Gradle will not check whether there is a module descriptor available for the dependency. In the next build file, we will see examples of the different artifact-only notations:

```
dependencies {
  // Using the @ext notation to specify
  // we only want the artifact for this
  // dependency.
  vehicles 'com.vehicles:car:2.0@jar'

  // Use map notation with ext attribute
  // to specify artifact only dependency.
  traffic group: 'com.traffic', name: 'pedestrian',
      version: '1.0', ext: 'jar'

  // Alternatively we can use the configuration closure.
  // We need to specify an artifact configuration closure
  // as well to define the ext attribute.
  vehicles('com.vehicles:car:2.0') {
    artifact {
      name = 'car-docs'
      type = 'tar'
      extension = 'tar'
    }
  }
}
```

A Maven module descriptor can use classifiers for the artifact. This is mostly used when a library with the same code is compiled for different Java versions, for example, a library is compiled for Java 5 and Java 6 with the `jdk15` and `jdk16` classifiers. We can use the `classifier` attribute when we define an external module dependency to specify which classifier we want to use. Also, we can use it in a string or map notation. With the string notation, we add an extra colon after the version attribute and specify the classifier. For the map notation, we can add the `classifier` attribute and specify the value we want. The following build file contains an example of the different definitions of a dependency with a classifier:

```
dependencies {
  // Using string notation we can
  // append the classifier after
  // the version attribute, prefixed
  // with a colon.
  vehicles 'com.vehicles:car:2.0:jdk15'

  // With the map notation we simply use the
  // classifier attribute name and the value.
  traffic group: 'com.traffic', name: 'pedestrian',
      version: '1.0', classifier: 'jdk16'

  // Alternatively we can use the configuration closure.
  // We need to specify an artifact configuration closure
  // as well to define the classifier attribute.
  vehicles('com.vehicles:truck:2.0') {
    artifact {
      name = 'truck'
      type = 'jar'
      classifier = 'jdk15'
    }
  }
}
```

In the following section, we will see how we can define client module dependencies in our build file.

Defining client module dependencies

When we define external module dependencies, we expect that there is a module descriptor file with information about the artifacts and dependencies for those artifacts. Gradle will parse this file and determine what needs to be downloaded. Remember that if such a file is not available on the artifact, it will be downloaded. However, what if we want to override the module descriptor or provide one if it is not available? In the module descriptor that we provide, we can define the dependencies of the module ourselves.

We can do this in Gradle with client module dependencies. Instead of relying on a module descriptor in a repository, we define our own module descriptor locally in the build file. We now have full control over what we think the module should look like and which dependencies the module itself has. We use the `module` method to define a client module dependency for a dependency configuration.

In the following example build file, we will write a client module dependency for the dependency car, and we will add a transitive dependency to the driver:

```
dependencies {
  // We use the module method to instruct
  // Gradle to not look for the module descriptor
  // in a repository, but use the one we have
  // defined in the build file.
  vehicles module('com.vehicles:car:2.0') {
    // Car depends on driver.
    dependency('com.traffic:driver:1.0')
  }
}
```

Using project dependencies

Projects can be part of a bigger, multi-project build, and the projects can be dependent on each other, for example, one project can be made dependent on the generated artifact of another project, including the transitive dependencies of the other project. To define such a dependency, we use the `project` method in our dependencies configuration block. We specify the name of the project as an argument. We can also define the name of a dependency configuration of the other project we depend on. By default, Gradle will look for the default dependency configuration, but with the `configuration` attribute, we can specify a specific dependency configuration to be used.

The next example build file will define project dependencies on the `car` and `truck` projects:

```
dependencies {
  // Use project method to define project
  // dependency on car project.
  vehicles project(':car')

  // Define project dependency on truck
  // and use dependency configuration api
  // from that project.
  vehicles project(':truck') {
    configuration = 'api'
  }

  // We can use alternative syntax
  // to specify a configuration.
  traffic project(path: ':pedestrian',
          configuration: 'lib')
}
```

Defining file dependencies

We can directly add files to a dependency configuration in Gradle. The files don't need to be stored in a repository but must be accessible from the project directory. Although most projects will have module descriptors stored in a repository, it is possible that a legacy project might have a dependency on files available on a shared network drive in the company. Otherwise, we must use a library in our project, which is simply not available in any repository. To add file dependencies to our dependency configuration, we specify a file collection with the `files` and `fileTree` methods. The following example build file shows the usage of all these methods:

```
dependencies {
  // Define a dependency on explicit file(s).
  vehicles files(
    'lib/vehicles/car-2.0.jar',
    'lib/vehicles/truck-1.0.jar'
  )

  // We can use the fileTree method to include
  // multiples from a directory and it's subdirectories.
  traffic fileTree(dir: 'deps', include: '*.jar')
}
```

The added files will not be part of the transitive dependencies of our project if we publish our project's artifacts to a repository, but they are if our project is part of a multi-project build.

Using internal Gradle and Groovy dependencies

When we write code to extend Gradle, such as custom tasks or plugins, we can have a dependency on the Gradle API and possibly the Groovy libraries used by the current Gradle version. We can use the `gradleApi` and `localGroovy` methods in our dependency configuration to have all the right dependencies.

If we are writing some Groovy code to extend Gradle, but we don't use any of the Gradle API classes, we can use `localGroovy`. With this method, the classes and libraries of the Groovy version shipped with the current Gradle version are added as dependencies. The following example build script uses the Groovy plugin and adds a dependency to the `compile` configuration on Groovy bundled with Gradle:

```
apply plugin: 'groovy'

dependencies {
  // Define dependency on Groovy
  // version shipped with Gradle.
  compile localGroovy()
}
```

When we write custom tasks or plugins for Gradle, we are dependent on the Gradle API. We need to import some of the API's classes in order to write our code. To define a dependency on the Gradle classes, we use the `gradleApi` method. This will include the dependencies for the Gradle version the build is executed for. The next example build file will show the use of this method:

```
apply plugin: 'groovy'

dependencies {
  // Define dependency on Gradle classes.
  compile gradleApi()
}
```

Using dynamic versions

Until now, we have set a version for a dependency explicitly with a complete version number. To set a minimum version number, we can use a special dynamic version syntax, for example, to set the dependency version to a minimum of 2.1 for a dependency, we use a version value of 2.1.+. Gradle will resolve the dependency to the latest version after version 2.1.0, or to version 2.1 itself. The upper bound is 2.2. In the following example, we will define a dependency on a spring-context version of at least 4.0.x:

```
dependencies {
  compile 'org.springframework:spring-context:4.0.+'
}
```

To reference the latest released version of a module, we can use `latest.integration` as the version value. We can also set the minimum and maximum version numbers we want. The following table shows the ranges we can use in Gradle:

Range	**Description**
`[1.0, 2.0]`	We can use all versions greater than or equal to 1.0 and lower than or equal to 2.0
`[1.0, 2.0[`	We can use all versions greater than or equal to 1.0 and lower than 2.0
`]1.0, 2.0]`	We can use all versions greater than 1.0 and lower than or equal to 2.0
`]1.0, 2.0[`	We can use all versions greater than 1.0 and lower than 2.0
`[1.0, )`	We can use all versions greater than or equal to 1.0
`]1.0, )`	We can use all versions greater than 1.0
`(, 2.0]`	We can use all versions lower than or equal to 2.0
`(, 2.0[`	We can use all versions lower than 2.0

In the following example build file, we will set the version for the spring-context module to greater than `4.0.1.RELEASE` and lower than `4.0.4.RELEASE`:

```
dependencies {
  // The dependency will resolve to version 4.0.3.RELEASE as
  // the latest version if available. Otherwise 4.0.2.RELEASE
  // or 4.0.1.RELEASE.
  compile 'org.springframework:spring-
  context:[4.0.1.RELEASE,4.0.4.RELEASE['
}
```

Getting information about dependencies

We have seen how we can define dependencies in our build scripts. To get more information about our dependencies, we can use the `dependencies` task. When we invoke the task, we can see which dependencies belong to the available configurations of our project. Also, any transitive dependencies are shown. The next example build file defines a dependency on Spring beans and we apply the Java plugin. We also specify a repository in the `repositories` configuration block. We will learn more about repositories in the next chapter. The following code captures the discussion in this paragraph:

```
apply plugin: 'java'

repositories {
  // Repository definition for JCenter Bintray.
  // Needed to download artifacts. Repository
```

```
      // definitions are covered later.
      jcenter()
    }

    dependencies {
      // Define dependency on spring-beans library.
      compile 'org.springframework:spring-beans:4.0.+'
    }
```

When we execute the `dependencies` task, we get the following output:

```
$ gradle -q dependencies

------------------------------------------------------------
Root project
------------------------------------------------------------

archives - Configuration for archive artifacts.
No dependencies

compile - Compile classpath for source set 'main'.
\--- org.springframework:spring-beans:4.0.+ -> 4.0.6.RELEASE
    \--- org.springframework:spring-core:4.0.6.RELEASE
          \--- commons-logging:commons-logging:1.1.3

default - Configuration for default artifacts.
\--- org.springframework:spring-beans:4.0.+ -> 4.0.6.RELEASE
    \--- org.springframework:spring-core:4.0.6.RELEASE
          \--- commons-logging:commons-logging:1.1.3

runtime - Runtime classpath for source set 'main'.
\--- org.springframework:spring-beans:4.0.+ -> 4.0.6.RELEASE
    \--- org.springframework:spring-core:4.0.6.RELEASE
          \--- commons-logging:commons-logging:1.1.3

testCompile - Compile classpath for source set 'test'.
\--- org.springframework:spring-beans:4.0.+ -> 4.0.6.RELEASE
    \--- org.springframework:spring-core:4.0.6.RELEASE
```

```
          \--- commons-logging:commons-logging:1.1.3

testRuntime - Runtime classpath for source set 'test'.
\--- org.springframework:spring-beans:4.0.+ -> 4.0.6.RELEASE
    \--- org.springframework:spring-core:4.0.6.RELEASE
          \--- commons-logging:commons-logging:1.1.3
```

We see all the configurations of our project, and for each configuration, we see the defined dependency with the transitive dependencies. Also, we can see how our dynamic version `4.0.+` is resolved to version `4.0.6.RELEASE`. To only see dependencies for a specific configuration, we can use the `--configuration` option for the `dependencies` task. We must use the value of the configuration we want to see the dependencies for. In the following output, we see the result when we only want to see the dependencies for the compile configuration:

```
$ gradle -q dependencies --configuration compile

------------------------------------------------------------
Root project
------------------------------------------------------------

compile - Compile classpath for source set 'main'.
\--- org.springframework:spring-beans:4.0.+ -> 4.0.6.RELEASE
    \--- org.springframework:spring-core:4.0.6.RELEASE
          \--- commons-logging:commons-logging:1.1.3
```

There is also the `dependencyInsight` incubating task in Gradle. Because it is incubating, the functionality or syntax can change in future versions of Gradle. With the `dependencyInsight` task, we can find out why a specific dependency is in our build and to which configuration it belongs. We have to use the `--dependency` option, the required one, with part of the name of the dependency. Gradle will look for dependencies where the group, name, or version contains part of the specified value for the `--dependency` option. Optionally, we can specify the `--configuration` option to only look for the dependency in the specified configuration. If we leave out this option, Gradle will look for the dependency in all the configurations of our project.

Let's invoke the `dependencyInsight` task to find the dependencies with Spring in the name and in the runtime configuration:

```
$ gradle -q dependencyInsight --dependency spring --configuration runtime
org.springframework:spring-beans:4.0.6.RELEASE

org.springframework:spring-beans:4.0.+ -> 4.0.6.RELEASE
```

```
\--- runtime

org.springframework:spring-core:4.0.6.RELEASE
\--- org.springframework:spring-beans:4.0.6.RELEASE
     \--- runtime
```

In the output, we see how version `4.0.+` is resolved to `4.0.6.RELEASE`. We also see that the `spring-beans` dependency and the transitive `spring-core` dependency are part of the runtime configuration.

Accessing dependencies

To access the configurations, we can use the `configurations` property of the Gradle project object. The `configurations` property contains a collection of `Configuration` objects. It is good to remember that a `Configuration` object is an instance of `FileCollection`. So, we can reference `Configuration` in our build scripts where `FileCollection` is allowed. The `Configuration` object contains more properties we can use to access the dependencies belonging to the configuration.

In the next example build, we will define two tasks that work with the files and information available from configurations in the project:

```
configurations {
  vehicles
  traffic.extendsFrom vehicles
}

task dependencyFiles << {
  // Loop through all files for the dependencies
  // for the traffic configuration, including
  // transitive dependencies.
  configurations.traffic.files.each { file ->
    println file.name
  }

  // We can also filter the files using
  // a closure. For example to only find the files
  // for dependencies with driver in the name.
  configurations.vehicles.files { dep ->
    if (dep.name.contains('driver')) {
      println dep.name
    }
  }

  // Get information about dependencies only belonging
```

```
    // to the vehicles configuration.
    configurations.vehicles.dependencies.each { dep ->
      println "${dep.group} / ${dep.name} / ${dep.version}"
    }

    // Get information about dependencies belonging
    // to the traffice configuration and
    // configurations it extends from.
    configurations.traffic.allDependencies.each {  dep ->
      println "${dep.group} / ${dep.name} / ${dep.version}"
    }
  }

  task copyDependencies(type: Copy) {
    description = 'Copy dependencies from configuration traffic to
    lib directory'

    // Configuration can be the source for a CopySpec.
    from configurations.traffic

    into "$buildDir/lib"
  }
```

Buildscript dependencies

When we define dependencies, we mostly want to define them for the code we are developing. However, we may also want to add a dependency to the Gradle build script itself. We can write code in our build files, which might be dependent on a library that is not included in the Gradle distribution. Let's suppose we want to use a class from the Apache Commons Lang library in our build script. We must add a `buildscript` configuration closure to our build script. Within the configuration closure, we can define repositories and dependencies. We must use the special `classpath` configuration to add dependencies to. Any dependency added to the `classpath` configuration can be used by the code in our build file.

Let's see how this works with an example build file. We want to use the `org.apache.commons.lang3.RandomStringUtils` class inside a `randomString` task. This class can be found in the `org.apache.commons:commons-lang3` dependency. We define this as an external dependency for the `classpath` configuration. We also include a repository definition inside the `buildscript` configuration block so that the dependency can be downloaded. The following code shows this:

```
  buildscript {
    repositories {
      // Bintray JCenter repository to download
      // dependency commons-lang3.
```

```
        jcenter()
      }

      dependencies {
        // Extend classpath of build script with
        // the classpath configuration.
        classpath 'org.apache.commons:commons-lang3:3.3.2'
      }
    }

    // We have add the commons-lang3 dependency
    // as a build script dependency so we can
    // reference classes for Apache Commons Lang.
    import org.apache.commons.lang3.RandomStringUtils

    task randomString << {
      // Use RandomStringUtils from Apache Commons Lang.
      String value = RandomStringUtils.randomAlphabetic(10)
      println value
    }
```

To include external plugins, which are not part of the Gradle distribution, we can also use the `classpath` configuration in the `buildscript` configuration block. In the next example build file, we will include the `Asciidoctor` Gradle plugin:

```
buildscript {
  repositories {
    // We need the repository definition, from
    // where the dependency can be downloaded.
    jcenter()
  }

  dependencies {
    // Define external module dependency for the Gradle
    // Asciidoctor plugin.
    classpath 'org.asciidoctor:asciidoctor-gradle-plugin:0.7.3'
  }
}

// We defined the dependency on this external
// Gradle plugin in the buildscript {...}
// configuration block
apply plugin: 'org.asciidoctor.gradle.asciidoctor'
```

Optional Ant task dependencies

We can reuse the existing Ant tasks in Gradle. The default tasks from Ant can be invoked from within our build scripts. However, if we want to use an optional Ant task, we must define a dependency for the classes needed by the optional Ant task. We create a new dependency configuration, and then we add a dependency to this new configuration. We can reference this configuration when setting the classpath for the optional Ant task.

Let's add the optional Ant task SCP for the secure copying of files to/from a remote server. We create the `sshAntTask` configuration to add dependencies for the optional Ant task. We can choose any name for the configuration. To define the optional task, we use the `taskdef` method from the internal `ant` object. The method takes a `classpath` attribute, which must be the actual path of all files of the `sshAntTask` dependencies. The `Configuration` class provides the `asPath` property to return the path to the files in a platform-specific way. So, if we use this on a Windows computer, the file path separator is a `;` and for other platforms it is a `:`. The following example build file contains all the code to define and uses the SCP Ant task:

```
configurations {
  // We define a new dependency configuration.
  // This configuration is used to assign
  // dependencies to, that are needed by the
  // optional Ant task scp.
  sshAntTask
}

repositories {
  // Repository definition to download dependencies.
  jcenter()
}

dependencies {
  // Define external module dependencies
  // for the scp Ant task.
  sshAntTask(group: 'org.apache.ant',
        name: 'ant-jsch',
        version: '1.9.4')
}

// New task that used Ant scp task.
task copyRemote(
  description: 'Secure copy files to remote server') << {

  // Define optional Ant task scp.
```

```
    ant.taskdef(
      name: 'scp',
      classname: 'org.apache.tools.ant.taskdefs.optional.ssh.Scp',

      // Set classpath based on dependencies assigned
      // to sshAntTask configuration. The asPath property
      // returns a platform-specific string value
      // with the dependency JAR files.
      classpath: configurations.sshAntTask.asPath)

    // Invoke scp task we just defined.
    ant.scp(
      todir: 'user@server:/home/user/upload',
      keyFile: '${user.home}/.ssh/id_rsa',
      passphrase: '***',
      verbose: true) {
      fileset(dir: 'html/files') {
        include name: '**/**'
      }
    }
  }
```

Managing dependencies

You have already learned earlier in the chapter that we can refactor the dependency definitions by extracting common parts into project properties. This way, we only have to change a few project property values to make changes to multiple dependencies.
In the next example build file, we will use lists to group dependencies together and reference those lists from the dependency definition:

```
  ext {
    // Group is used multiple times, so
    // we extra the variable for re-use.
    def vehiclesGroup = 'com.vehicles'

    // libs will be available from within
    // the Gradle script code, like dependencies {...}.
    libs = [
      vehicles: [
        [group: vehiclesGroup, name: 'car', version: '2.0'],
        [group: vehiclesGroup, name: 'truck', version: '1.0']
      ],
      traffic: [
        [group: 'com.traffic', name: 'pedestrian', version:
        '1.0']
      ]
    ]
  }

  configurations {
    vehicles
  }

  dependencies {
    // Reference ext.libs.vehicles defined earlier
    // in the build script.
    vehicles libs.vehicles
  }
```

Maven has a feature called dependency management metadata that allows us to define versions used for dependencies in a common part of the build file. Then, when the actual dependency is configured, we can leave out the version because it will be determined from the dependency management section of the build file. Gradle doesn't have such a built-in feature, but as illustrated earlier, we can use simple code refactoring to get a similar effect.

We can still have declarative dependency management, as we do in Maven, in our Gradle build, with the external dependency management plugin by Spring. This plugin adds a `dependencyManagement` configuration block to Gradle. Inside the configuration block, we can define dependency metadata, such as the group, name, and version. In the `dependencies` configuration closure in our Gradle build script, we don't have to specify the version anymore because it will be resolved via the dependency metadata in the `dependencyManagement` configuration. The following example build file uses this plugin and specifies dependency metadata using `dependencyManagement`:

```
buildscript {
  repositories {
    // Specific repository to find and download
    // dependency-management-plugin.
    maven {
      url 'http://repo.spring.io/plugins-snapshot'
    }
  }
  dependencies {
    // Define external module dependency with plugin.
    classpath 'io.spring.gradle:dependency-management-
    plugin:0.1.0.RELEASE'
  }
}

// Apply the external plugin dependency-management.
apply plugin: 'io.spring.dependency-management'
apply plugin: 'java'

repositories {
  // Repository for downloading dependencies.
  jcenter()
}

// This block is added by the dependency-management
// plugin to define dependency metadata.
dependencyManagement {
  dependencies {
    // Specify group:name followed by required version.
    'org.springframework.boot:spring-boot-starter-web'
    '1.1.5.RELEASE'

    // If we have multiple module names for the same group
    // and version we can use dependencySet.
    dependencySet(group: 'org.springframework.boot',
          version: '1.1.5.RELEASE') {
      entry 'spring-boot-starter-web'
      entry 'spring-boot-starter-actuator'
    }
  }
}

dependencies {
  // Version is resolved via dependencies metadata
  // defined in dependencyManagement.
  compile 'org.springframework.boot:spring-boot-starter-web'
}
```

To import a Maven **bill of materials** (**BOM**) provided by an organization, we can use the `imports` method inside the `dependencyManagement` configuration. In the next example, we will use the Spring IO platform BOM. In the `dependencies` configuration, we can leave out the version because it will be resolved via the BOM:

```
  buildscript {
    repositories {
      // Specific repository to find and download
      // dependency-management-plugin.
      maven {
        url 'http://repo.spring.io/plugins-snapshot'
      }
    }
    dependencies {
      // Define external module dependency with plugin.
      classpath 'io.spring.gradle:dependency-management-
      plugin:0.1.0.RELEASE'
    }
  }

  // Apply the external plugin dependency-management.
  apply plugin: 'io.spring.dependency-management'
  apply plugin: 'java'

  repositories {
    // Repository for downloading BOM and dependencies.
    jcenter()
  }

  // This block is added by the dependency-management
  // plugin to define dependency metadata.
  dependencyManagement {
    imports {
      // Use Maven BOM provided by Spring IO platform.
      mavenBom 'io.spring.platform:platform-bom:1.0.1.RELEASE'
    }
  }

  dependencies {
    // Version is resolved via Maven BOM.
    compile 'org.springframework.boot:spring-boot-starter-web'
  }
```

Summary

In this chapter, you learned how to create and use dependency configurations to group together dependencies. We saw how to define several types of dependencies, such as external module dependency and internal dependencies.

Also, we saw how we can add dependencies to code in Gradle build scripts with the `classpath` configuration and the `buildscript` configuration.

Finally, we looked at some maintainable ways of defining dependencies using code refactoring and the external dependency management plugin.

In the next chapter, we will learn more about how we can configure repositories that store dependency modules.