Why Build Your Java Projects with Gradle Rather than Ant or Maven?

drdobbs.com/jvm/why-build-your-java-projects-with-gradle/240168608

Why Build Your Java Projects with Gradle Rather than Ant or Maven? By Benjamin Muschko , July 08, 2014 4 Comments The default build tool for Android (and the new star of build tools on the JVM) is designed to ease scripting of complex, multi-language builds. Should you change to it, though, if you're using Ant or Maven?

JVM Languages

For years, builds had the simple requirements of compiling and packaging software. But the landscape of modern software development has changed, and so have the needs for build automation. Today, projects involve large and diverse software stacks, incorporate multiple programming languages, and apply a broad spectrum of testing strategies. With the rise of agile practices, builds must support early integration of code as well as frequent and easy delivery to both test and production environments.

Established build tools regularly fall short in meeting these goals. How many times have your eyes glazed over while looking at XML to figure out how a build works? And why can't it be easier to add custom logic to your build? All too often, when adding on to a build script, you can't shake the feeling that you're implementing a workaround or hack. I feel your pain. There has to be a better way of doing these things in an expressive and maintainable way. There is — it's called Gradle.

Gradle is the next evolutionary step in JVM-based build tools. It draws on lessons learned from established tools such as Ant and Maven and takes their best ideas to the next level. Following a build-by-convention approach, Gradle allows for declaratively modeling your problem domain using a powerful and expressive domain-specific language (DSL) implemented in Groovy instead of XML. Because Gradle is a JVM native, it allows you to write custom logic in the language you're most comfortable with, be it Java or Groovy.

In the Java world, a remarkably large number of libraries and frameworks are used. Dependency management is employed to automatically download these artifacts from a repository and make them available to your application. Having learned from the shortcomings of existing

dependency management solutions, Gradle provides its own implementation. Not only is it highly configurable, it also strives to be as compatible as possible with existing dependency management infrastructures (such as Maven and Ivy). Gradle's ability to manage dependencies isn't limited to external libraries. As your project grows in size and complexity, you'll want to organize the code into modules with clearly defined responsibilities. Gradle provides powerful support for defining and organizing multiproject builds, as well as modeling dependencies between projects.

To get started with Gradle, all you need to bring to the table is a good understanding of the Java programming language. If you're new to project automation or haven't used a build tool before, my book *Gradle in Action* is a good place to start.

In this two-part series on using Gradle, I first compare existing JVM-language build tools with the features Gradle has to offer and describe Gradle's compelling feature set. In the next installment, I cover installing Gradle and writing and executing a simple Gradle script.

Why Gradle? Why Now?

If you've ever dealt with build systems, frustration may be one of the feelings that comes up when thinking about the challenges you've faced. Shouldn't the build tool naturally help you accomplish the goal of automating your project? Instead, you had to compromise on maintainability, usability, flexibility, extendibility, or performance.

Let's say you want to copy a file to a specific location when you're building the release version of your project. To identify the version, you check a string in the metadata describing your project. If it matches a specific numbering scheme (for example, 1.0-RELEASE), you copy the file from point A to point B. From an outside perspective, this may sound like a trivial task. If you have to rely on XML, the build language of many traditional tools, expressing this simple logic becomes fairly difficult. The build tool's response is to add scripting functionality through nonstandard extension mechanisms. You end up mixing scripting code

with XML or invoking external scripts from your build logic. It's easy to imagine that you'll need to add more and more custom code over time. As a result, you inevitably introduce accidental complexity, and maintainability goes out the window. Wouldn't it make sense to use an expressive language to define your build logic in the first place?

Here's another example. Maven follows the paradigm of convention-over-configuration by introducing a standardized project layout and build lifecycle for Java projects. That's a great approach if you want to ensure a unified application structure for a green-field project — a project that lacks any constraints imposed by prior work. However, you may be the lucky one who needs to work on one of the many legacy projects that use different conventions. One of the conventions Maven is very strict about is that one project needs to produce one artifact, such as a JAR file. But how do you create two different JAR files from one source tree without having to change your project structure? Just for this purpose, you'd have to create two separate projects. Again, even though you can make this happen with a workaround, you can't shake off the feeling that your build process will need to adapt to the tool, not the tool to your build process.

These are only some of the issues you may have encountered with existing solutions. Often you've had to sacrifice nonfunctional requirements to model your enterprise's automation domain. But enough with the negativity — let's see how Gradle fits into the build tool landscape.

Evolution of Java Build Tools

Let's look at how build tools have evolved over the years. Two tools have dominated building Java projects: Ant and Maven. Over the course of years, both tools improved significantly and extended their feature set. But even though both are highly popular and have become industry standards, they have one weak point: build logic has to be described in XML. XML is great for describing hierarchical data, but falls short on expressing program flow and conditional logic. As a build script grows in complexity, maintaining the build code becomes a nightmare.

Ant's first official version was released in 2000. Each element of work (a *target* in Ant's lingo) can be combined and reused. Multiple targets can be chained to combine single units of work into full workflows. For example, you might have one target for compiling Java source code and another one for creating a JAR file that packages the class files. Building a JAR file only makes sense if you first compiled the source code. In Ant, you make the JAR target depend on the compile target. Ant doesn't give any guidance on how to structure your project. Though it allows for maximum flexibility, Ant makes each build script unique and hard to understand. External libraries required by your project are usually checked into version control, because there is no automated mechanism to pull them from a central location. Early versions of Ant required a lot of discipline to avoid repetitive code. Its extension mechanism was simply too weak. As a result, the bad coding practice of copying and pasting code was the only viable option. To unify project layouts, enterprises needed to impose standards.

Maven 1, released in July 2004, tried to ease that process. It provided a standardized project and directory structure, as well as dependency management. Unfortunately, custom logic is hard to implement. If you want to break out of Maven's conventions, writing a plugin, called a *Mojo*, is usually the only solution. The name Mojo might imply a straightforward, easy, and sexy way to extend Maven; in reality, writing a plugin in Maven is cumbersome and overly complex.

Later, Ant caught up with Maven by introducing dependency management through the Apache library Ivy, which can be fully integrated with Ant to declaratively specify dependencies needed for your project's compilation and packaging process. Maven's dependency manager, as well as Ivy, support resolving transitive dependencies. When I speak of transitive dependencies, I mean the graph of libraries required by your specified dependencies. A typical example of a transitive dependency would be the XML parser library Xerces that requires the XML APIs library to function correctly. Maven 2, released in October 2005,

took the idea of convention over configuration even further. Projects consisting of multiple modules could define their dependencies on each other.

These days a lot of people are looking for alternatives to established build tools. We see a shift from using XML to a more expressive and readable language to define builds. A build tool that carries on this idea is Gant, a DSL on top of Ant written in Groovy. Using Gant, users can now combine Groovy features with their existing knowledge of Ant without having to write XML. Even though it wasn't part of the core Maven project, a similar approach was proposed by the project Maven Polyglot that enables you to write your build definition logic, which is the project object model (POM) file, in Groovy, Ruby, Scala, or Clojure.

We're on the cusp of a new era of application development: polyglot programming. Many applications today incorporate multiple programming languages, each of which is best suited to implement a specific problem domain. It's not uncommon to face projects that use client-side languages like JavaScript that communicate with a mixed, multilingual back end like Java, Groovy, and Scala, which in turn makes calls to a C++ legacy application. It's all about the right tool for the job. Despite the benefits of combining multiple programming languages, your build tool needs to fluently support this infrastructure as well. JavaScript needs to be merged, minified, and zipped, and your server-side and legacy code needs to be compiled, packaged, and deployed.

Gradle fits right into that generation of build tools and satisfies many requirements of modern build tools (Figure 1). It provides an expressive DSL, a convention over configuration approach, and powerful dependency management. It makes the right move to abandon XML and introduce the dynamic language Groovy to define your build logic. Sounds compelling, doesn't it?

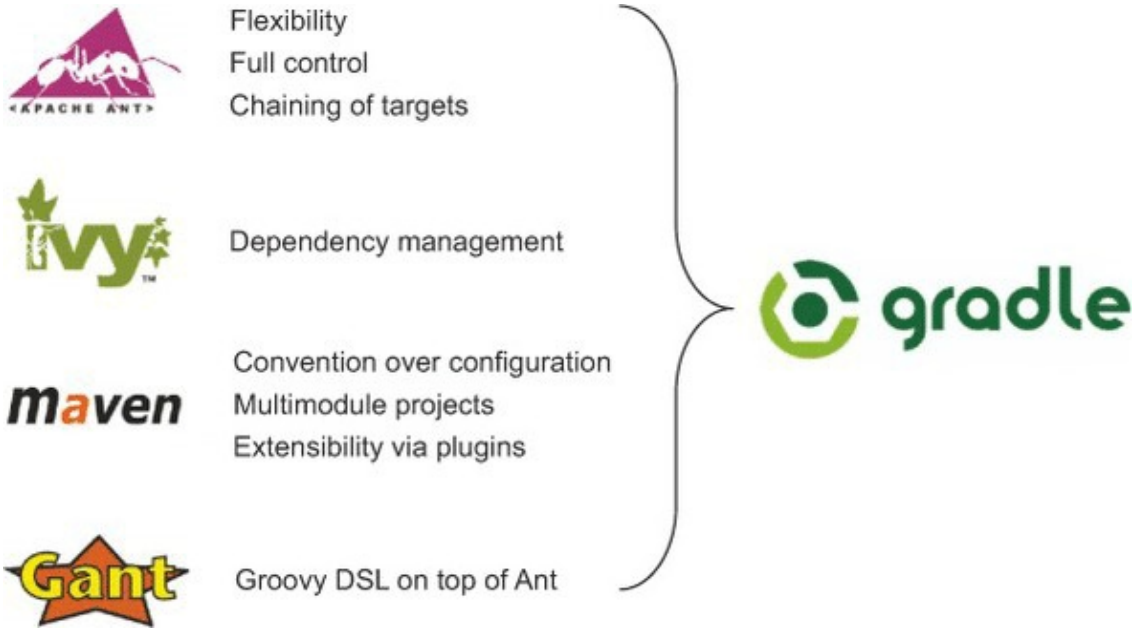

Figure 1: Gradle combines the best features from other build tools.

Why Choose Gradle?

If you're a developer, automating your project is part of your day-to-day business. Don't you want to treat your build code like any other piece of software that can be extended, tested, and maintained? Let's put software engineering back into the build. Gradle build scripts are declarative, readable, and clearly express their intention. Writing code in Groovy instead of XML, sprinkled with Gradle's build-by-convention philosophy, significantly cuts down the size of a build script and is far more readable (see Figure 2).

Maven

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.mycompany.app</groupId>
   <artifactId>my-app</artifactId>
   <packaging>jar</packaging>
   <version>1.0-SNAPSHOT</version>

   <dependencies>
      <dependency>
         <groupId>junit</groupId>
         <artifactId>junit</artifactId>
         <version>4.11</version>
         <scope>test</scope>
      </dependency>
   </dependencies>
</project>
```

Gradle

```
apply plugin: 'java'
group = 'com.mycompany.app'
archivesBaseName = 'my-app'
version = '1.0-SNAPSHOT'

repositories {
   mavenCentral()
}

dependencies {
   testCompile 'junit:junit:4.11'
}
```

Figure 2: Comparing build script size and readability between Maven and Gradle.

It's impressive to see how much less code you need to write in Gradle to achieve the same goal. With Gradle you don't have to make compromises. Where other build tools like Maven propose project layouts that are "my way or the highway," Gradle's DSL allows for flexibility by adapting to nonconventional project structures.

Never change a running system, you say? Your team already spent a lot of time on establishing your project's build code infrastructure. Gradle doesn't force you to fully migrate all of your existing build logic. Good integration with other tools like Ant and Maven is at the top of Gradle's priority list.

The market seems to be taking notice of Gradle. Popular open source projects like Groovy and Hibernate completely switched to Gradle as the backbone for their builds. Every Android project ships with Gradle as the default build system. Gradle also had an impact on the commercial market. Companies like Orbitz, EADS, and Software AG embraced Gradle as well, to name just a few. VMware, the company behind Spring and Grails, made significant investments in choosing Gradle. Many of their software products, such as the Spring framework and Grails, are literally built on the trust that Gradle can deliver.

Gradle's Compelling Feature Set

Let's take a closer look at what sets Gradle apart from its competitors: its compelling feature set (see Figure 3). To summarize, Gradle is an enterprise-ready build system, powered by a declarative and expressive Groovy DSL. It combines flexibility and effortless extendibility with the idea of convention over configuration and support for traditional dependency management. Backed by a professional services company (Gradleware) and strong community involvement, Gradle is becoming the number-one choice build solution for many open source projects and enterprises.

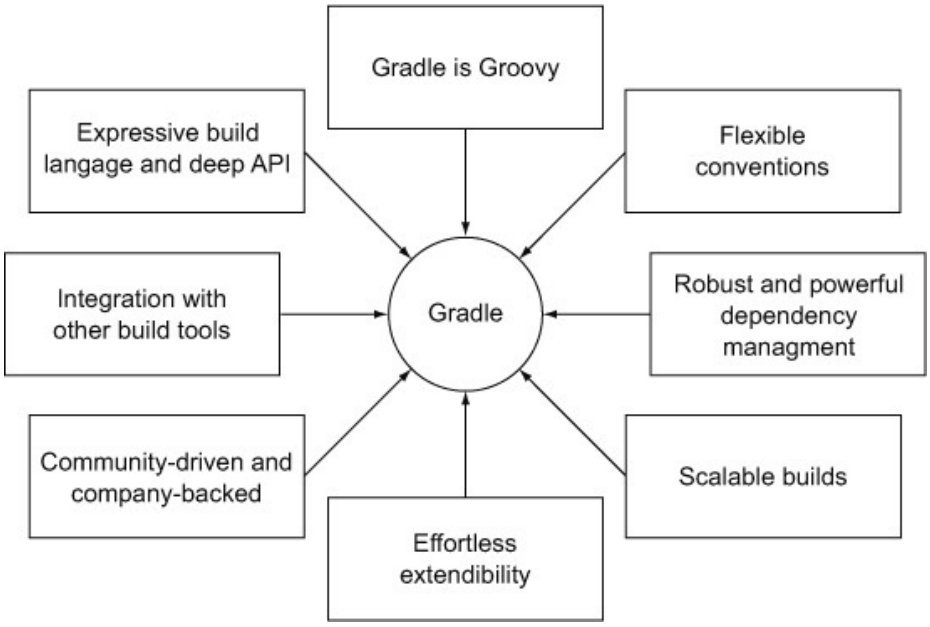

Figure 3: Gradle's compelling feature set.