

JAVA BUILD TOOLS: PART 1

AN INTRODUCTORY CRASH COURSE TO GETTING
STARTED WITH MAVEN, GRADLE AND ANT + IVY

↖
*Start building
tomorrow's apps yesterday!*



TABLE OF CONTENTS

CHAPTER I

AN INTRODUCTION AND [RATHER] SHORT HISTORY
OF BUILD TOOLS

1-10

CHAPTER II

HOW TO GET SET UP WITH MAVEN, GRADLE AND ANT + IVY

GOING FROM ZERO TO HERO WITH MAVEN

12-16

GOING FROM ZERO TO HERO WITH GRADLE

17-19

GOING FROM ZERO TO HERO WITH ANT + IVY

20-25

CHAPTER III

GETTING YOUR HANDS DIRTY WITH PLUGINS, EXTENSIONS
AND CUSTOMIZATIONS

26-32

CHAPTER IV

SUMMARY, CONCLUSION AND GOODBYE COMIC :-)

33-37

CHAPTER 1

AN INTRODUCTION AND [RATHER] SHORT HISTORY OF BUILD TOOLS

Build tools are an integral part of what makes our lives easier between checking code in and testing your product. Love them or hate them, they're here to stay so let's first take a look at what they are, how they emerged and why they are both hated and revered today.

What is a build tool and what does it do?

Let's start at the beginning. What is a build tool? Well, as its name suggests, it is a tool for building - shocker! OK, that's a bit too simple: **build tools are primarily used to compile and construct some form of usable software image from your source code.** This software image might be a web application, a desktop application, a library for other code bases to use or even a full product.

Build tools have evolved over the years, becoming progressively more sophisticated and feature rich. This provides developers with useful additions, such as the ability to manage your project dependencies as well as automate tasks beyond compiling and packaging. Typically, build tools require two major components: a **build script** and an **executable**, which processes the build script. Build scripts can be (and should be these days!) platform agnostic, eg. it may be executed on Windows, Linux or Mac at the same time.

We broke it down into this reasonable list of main tasks and requirements a good build tool should be good at:

- Managing dependencies
- Incremental compilation
- Properly handling compilation and resource management tasks
- Handling different profiles (development vs. production)
- Adapting to changing product requirements
- And last but not least: designed to automate builds

Tools we'll cover in this report: Maven, Gradle and Ant + Ivy

Although it's not possible to give proper coverage to all the build tools in the world, we hope that by covering the top 3--Maven, Gradle and Ant + Ivy--in relative detail will be enough for most readers. If not, feel free to introduce us to some new technology!

Firstly, Apache Maven. Maven is a Yiddish word meaning accumulator of knowledge. It is currently the most popular (by number of users) build tool on the market today, and often the *de facto* first choice, among Java developers. Originally, Maven was created in 2002 but really struck a chord with Java developers in 2005 when Maven 2 was released.

The logo for Apache Maven, featuring the word "maven" in a bold, italicized sans-serif font. The letter "a" is orange, while the other letters are black.

We'll also look at the combination of Ant and Ivy in partnership. Both, projects from Apache, bring different things to the build tool table. Ant provides the capabilities to run specific tasks or targets, such as compile, test, build etc, whereas Ivy gives the dependency management required to enable larger more complex projects with dependency trees to be managed in a much easier way.



Lastly, but by no means least, we will also be looking at Gradle. As the newest build tool that we will focus on in our report (version 1 released in 2012), Gradle is a tool designed around multi-project environments. It takes a lot from what was learned from Maven and Ant, while daring to tread new ground, like choosing to use a Groovy style DSL rather than XML for its config scripts. Gradle's plugins primarily focuses on Java, Groovy and Scala.



Other build tools which are worth mentioning and looking at include SBT, Tesla, Buildr, Tweaker, Leiningen and of course Make! However, one of these we feel like should have a bit of a call out going to talk about in more detail...

Managing dependencies seems pretty important. Why is that?

One big area of pain which build tools have taken away/eased is dependency management. What did we do before this was integrated into a build tool? Ha! Do you remember importing a library into your environment and physically checking the binaries into your code repository? Or adding a script to download your external source dependencies hoping that URL remained a constant! Let's not even think about the dependencies your dependencies have, or their dependencies or... oh no, I've gone cross-eyed again!

Fear not, this is more a description of the past than the present or future. We now live in a world where it's much easier to decipher the spaghetti of dependencies with build tools, making use of the more modular software world we live in. That's not to say it's all perfect and there aren't any more problems we face today with dependencies, but it has certainly come a long way.

Let's break this down further and look at the areas that could fall under the dependency management heading:

- Build dependency
- External dependencies
- Multi-module projects
- Repositories
- Plugins
- Publishing artifacts

The three main build tools we look at in this report all have their own ways of handling dependencies as well as their differences. However, one thing is consistent: each module has an identity which consists of a group of some kind, a name and a version. In a more complex project of course, many different modules may have similar dependencies at different versions or version ranges. This can often cause problems, design meetings and headaches.

Build tools allow us to specify a version range for our dependencies instead of a specific version if we so wish. This allows us to find a common version of a dependency across multiple modules that have requirements. These can be extremely useful and equally dangerous if not used sensibly. If you've used version ranges before, you'll likely have used a similar if not identical syntax in the past. Here's the version range lowdown:

```
[x,y] - from version x up to version y, inclusive
(x,y) - from version x up to version y, exclusive
[x,y) - from version x, inclusive up to version y, exclusive
(x,y] - from version x, exclusive up to version y, inclusive
[x,) - from version x inclusive and up!
(,x] - from version x inclusive and down!
[x,y),(y,) - from version x inclusive and up, but excluding
version y, specifically!
```

While each build tool respects this range format, when modules have conflicting ranges the build tools can act differently. Let's take an example with Google Guava and see how each build tool reacts. For instance, consider the following example in which we have two modules in the same project each depending on the same module by name, but with differing version ranges:

```
Module A
  com.google.guave:guava:[11.0,12.99]
Module B
  com.google.guave:guava:[13.0,)
```

Here, module A requires a Guava module between the version range 11.0 to 12.99 inclusive, which is mutually exclusive to the version range of guava Module B requires, which is anything over and including version 13.0. So what do our build tools, Maven, Gradle and Ant + Ivy, do in this situation?

Well, Maven will result in an error as there is no version available which satisfies both ranges, whereas Gradle decides to pick the highest range available in either of the ranges, which means version 15.0 (The highest Guava version available at the time of writing). But, but, but that's outside the range Module A has explicitly described...right? We'll let you decide :) What does Ivy (Ant's dependency management enabling addition) do here? Well, it's very similar in it's behavior to Gradle, which isn't surprising given Gradle once used Ivy as its underlying implementation for dependency management.

We wrote more on this subject in a blog post:

<http://zeroturnaround.com/rebellabs/java-build-tools-how-dependency-management-works-with-maven-gradle-and-ant-ivy/>

Weigh in on the DSL vs. XML debate

As build tools have evolved into different features and flavors, a debate has arisen over the topic of virtues--using good old XML, introduced in 1996, versus DSL (domain specific language) commands for your build tool.

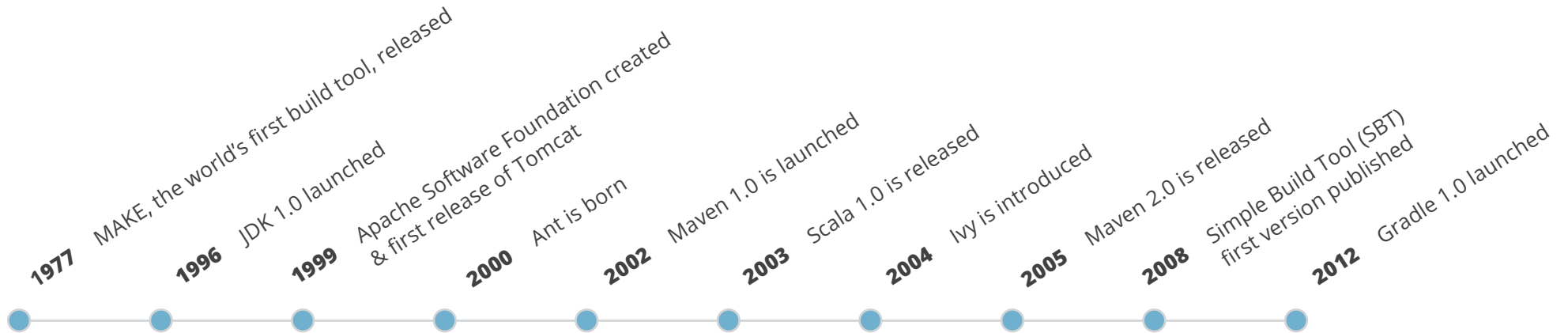
One of the benefits of using XML-based tools like Ant and Maven is that XML is strictly structured and highly standardized, which means there is only one way to write, and one way to read, XML. Therefore, XML is easily parseable and interpretable by automatic tools, and it's not necessary to create some special software to handle it, since there are so many of them out there that handle it pretty well.

DSL-based tools allow the user to enjoy a much higher level of customization. Gradle is based in Groovy, which is loosely related to Java and understandable with a small learning curve. Using Gradle, you don't have to write plugins or obscure bunches of XML; rather, you write stuff right inside the script.

From a lines of code (LoC) perspective, DSL is considerably less verbose and closer to the problem domain--one of the biggest complaints about XML is that there is too much boilerplate (i.e. our Ant build script is [69 lines of code](#), for example), whereas you can start Gradle with a single line of code. The drawback here is that while XML is very long and transparent, DSL is not very straightforward to interpret and it's hard to see what's going on under the covers.

THE EVOLUTION OF BUILD TOOLS: 1977 - 2013 (AND BEYOND)

Visual timeline



Did you know that Ant was originally written as a build tool for Apache Tomcat, when Tomcat was open-sourced? Before that, there was no platform-independent build tool.

Did you know that Maven was originally written to be a build tool for Apache Turbine web framework? Turbine is long forgotten, but the way Maven handles dependencies has become *de-facto* standard for dependency management in Java.

Build tool popularity from 2010 - 2013

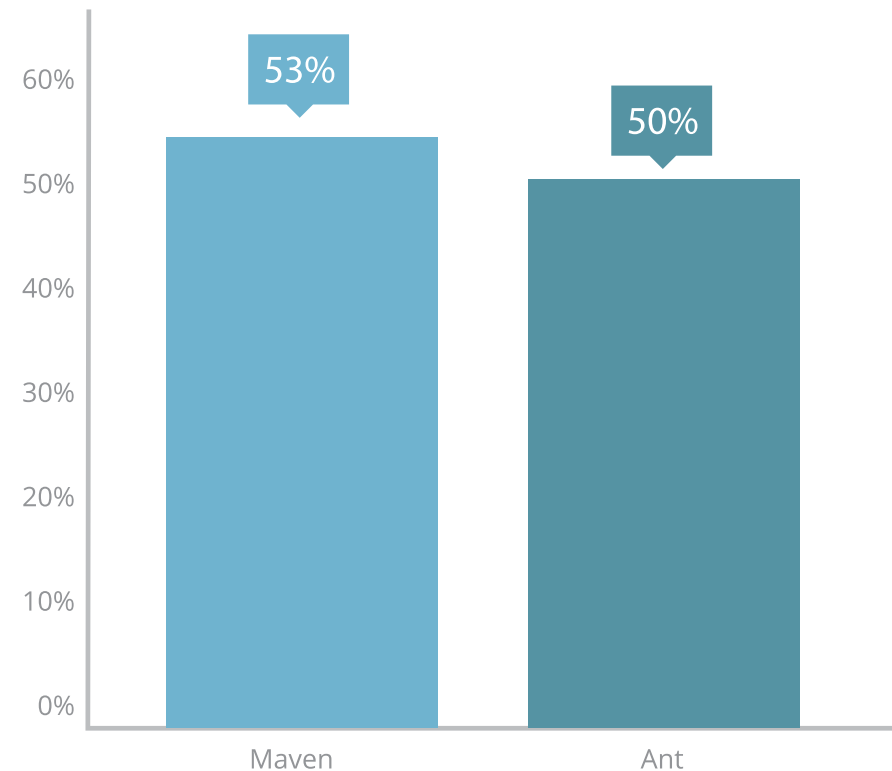
We can't cover all the build tools the world has ever seen, so in this report we have focused our efforts on the big three build tools which people really use in anger today for hobby projects all the way up to full scale enterprise environments.

One cool thing is that we've been looking at Build Tools in use since 2010, and we've seen the following trends over the last few years. Here are the self-reported statistics from 3 years worth of developer data.

Note: In surveys like this some bias is always present. In many cases, developers used more than one build tool in different projects, so these results are designed to be *indicative of market share / popularity*, not a precise measurement based on exclusive responses.

To look at the trends, we can see that Ant (with or without Ivy) is losing ground. Maven is at the stable "market leader" position, slightly increasing share depending on who is asked. Gradle, which wasn't available until 2012, jumped into the scene and started grabbing users, potentially encouraging enough Ant + Ivy users to jump ship while others maintained consistency with Maven.

Build Tools Popularity - Late 2010

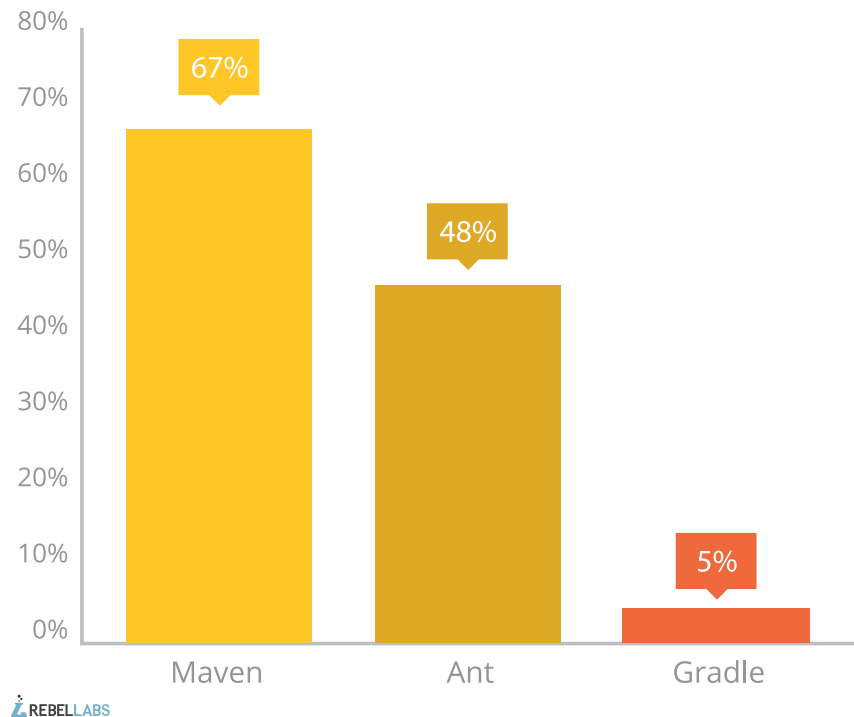


Sample size: 1000+ developers

Source: Java EE Productivity Report 2011 © ZeroTurnaround

<http://zeroturnaround.com/rebellabs/java-ee-productivity-report-2011/>

Build Tools Popularity - Early 2012

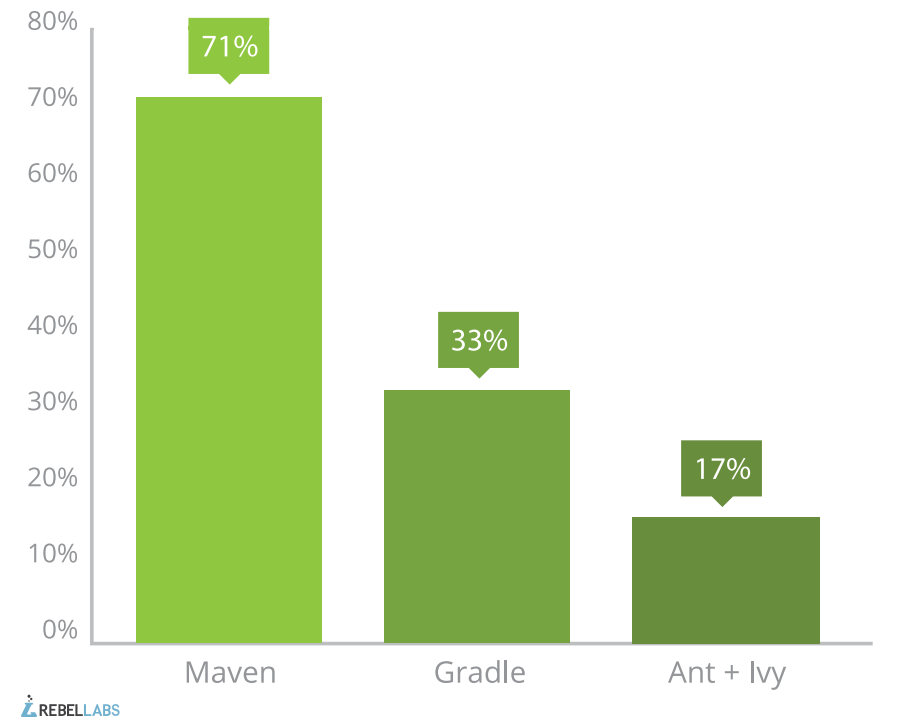


Sample size: 1000+ developers

Source: Productivity Report 2012 © ZeroTurnaround

<http://zeroturnaround.com/rebellabs/developer-productivity-report-2012-java-tools-tech-devs-and-data/>

Build Tools Popularity - Mid 2013



Sample size: 780 developers

Source: Code Impossible (a blog by JRebel Product Manager

Anton Arhipov)

<http://arhipov.blogspot.com/2013/08/java-build-tools-survey-results.html>

Special callout to Simple Build Tool (SBT)

We felt that SBT, with its powerful capabilities and small but strong community, deserves a special section. We aren't going to cover SBT in this report, as the number of developers using it (mainly for Scala development) isn't really high enough, but SBT has a lot to offer and some think that it has the potential to be a real go-to tool for JVM development.

At a first glance, one could think it similar to Gradle as they've both taken the DSL approach, but the important difference lies in static typing. The build management script is raised to the same standard as application code, and it really shows. Browsing documentation gets replaced with auto-completion, and errors surface before execution.

The documentation tends to be too academically correct for our tastes, but once you've gotten used to it, you'll notice that there's only one concept to learn--settings. Everything in SBT is represented by a setting--a pair with a key and a value. And by everything, we really mean everything, from the current version string to tasks with complex dependency trees.

SBT's simple core makes it easy to understand and control what exactly is going on with your project. At the same time, as any behavior is just a list of settings, it's trivial to externalize and share it as a plugin. This is strongly illustrated by the fact that while the community is not as large as say Maven's, we haven't yet started a search for a plugin and come out empty handed.

So the next time you're in an adventurous mood, take it for a test drive and maybe you'll be so impressed, that everything else just doesn't seem to cut it.

CHAPTER II:

HOW TO GET SET UP WITH MAVEN, GRADLE AND ANT + IVY

If you're a RebelLabs regular reader, you can guess what comes here. The section for beginners to get started and set up with their new tool--from download and installation to sharing artifacts to assumptions and restrictions. If you're an existing user of Maven, Gradle or Ant and would like to see more complex stuff, skip ahead to Chapter III, where we go all bit more ninja.

Going from zero to hero with Maven

Ok, let's start from the very beginning. Before using Maven, you must have Java installed, and that's about it...now we'll show you a few different ways to get Maven up and running.

When you unpack the archive, you may also add the `<maven_install_dir>/bin` to PATH, if you don't want to write down the full path to the executable every time you need to run a `mvn` command.

INSTALLING MAVEN FROM THE WEBSITE

You can download Maven from the Apache Software Foundation website (<http://maven.apache.org/download.cgi>). It's a "one archive fits all" operation and you can also choose .zip or tar.gz.

Download Apache Maven 3.1.1

Maven is distributed in several formats for your convenience. Use a source archive if you intend to build Maven yourself. Otherwise, simply pick a ready-made binary distribution and follow the installation instructions given at the end of this document.

You will be prompted for a mirror - if the file is not found on yours, please be patient, as it may take 24 hours to reach all mirrors.

In order to guard against corrupted downloads/installations, it is highly recommended to [verify the signature](#) of the release bundles against the public [KEYS](#) used by the Apache Maven developers.

Maven is distributed under the [Apache License, version 2.0](#).

We **strongly** encourage our users to configure a Maven repository mirror closer to their location, please read [How to Use Mirrors for Repositories](#).

Be sure to check the [compatibility notes](#) before using this version to avoid surprises. While Maven 3 aims to be backward-compatible with Maven 2.x to the extent possible, there are still a few significant changes.

Mirror

The currently selected mirror is <http://servingzone.com/mirrors/apache/>. If you encounter a problem with this mirror, please select another mirror. If all mirrors are failing, there are *backup* mirrors (at the end of the mirrors list) that should be available.

Other mirrors:

You may also consult the [complete list of mirrors](#).

Maven 3.1.1

This is the current stable version of Maven.

	Link	Checksum	Signature
Maven 3.1.1 (Binary tar.gz)	apache-maven-3.1.1-bin.tar.gz	apache-maven-3.1.1-bin.tar.gz.md5	apache-maven-3.1.1-bin.tar.gz.asc
Maven 3.1.1 (Binary zip)	apache-maven-3.1.1-bin.zip	apache-maven-3.1.1-bin.zip.md5	apache-maven-3.1.1-bin.zip.asc
Maven 3.1.1 (Source tar.gz)	apache-maven-3.1.1-src.tar.gz	apache-maven-3.1.1-src.tar.gz.md5	apache-maven-3.1.1-src.tar.gz.asc
Maven 3.1.1 (Source zip)	apache-maven-3.1.1-src.zip	apache-maven-3.1.1-src.zip.md5	apache-maven-3.1.1-src.zip.asc
Release Notes	3.1.1		
Release Reference Documentation	3.1.1		

INSTALLING MAVEN THROUGH CLI

Some operating systems like Linux and Mac OS already have Maven in their package repositories, so you can install it from there. For example, the command for Ubuntu (and other Debian-based linux distributives) will look like this:

```
sudo apt-get install maven
```

One concern here is that your OS package managers often have older versions compared to what's available from the download site, so check there to make sure you are downloading the latest version.

INSTALLING MAVEN FROM IDES

Most IDEs, including Eclipse, IntelliJ IDEA and NetBeans, support Maven out of the box. They also allow you to seamlessly import your Maven projects and help you managing it with features like auto-complete and automatic adding/removing dependencies based on your imports and other things.

WRITING THE BUILD SCRIPT FOR MAVEN

For showing you the build script, we'll use the well-known [Pet Clinic application](#) based on the Spring Framework to demonstrate you the principles of working with Maven. Clone it somewhere on your machine with this command:

```
git clone https://github.com/spring-projects/spring-petclinic.git
```

Now change the directory to the one you have just cloned the project into. There should be a **pom.xml** file. We will refer `<project_root>` to this folder from now on. The Pet Clinic sample app is a Maven application,

meaning it follows Maven conventions and already has a Maven build script, so we don't have to write it. Nevertheless we still will go through **pom.xml** to understand, how it is written.

The first 9 lines of build script tell us basic things about the project.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" [...] >
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.springframework.samples</groupId>
  <artifactId>spring-petclinic</artifactId>
  <version>1.0.0-SNAPSHOT</version>

  <name>petclinic</name>
  <packaging>war</packaging>
```

Let's break this down a bit here:

- The top-level element in every Maven build script is *project*.
- The *modelVersion* tag shows which version of object model current **pom.xml** uses.
- *groupId* is a unique identifier of the organization or group that created this project. Most often it is based on the fully-qualified domain name that belongs to the organization.
- *ArtifactId* is the base name of the artifact created by the project.
- *Version* is pretty self-explanatory.
- *Name* is just a project title, which is often used in generated documentation.
- *Packaging* shows the type of the artifact (jar, war, ear) to be built and it also adds some specific lifecycle tasks that can be run for particular packaging.

The rest of the build script can be visually divided into 3 big parts: **properties**, **dependencies** (including *dependencyManagement* tag) and **build**.

Properties can be considered as a declaration of constants that can be used later in the script. For example,

```
<spring-framework.version>3.2.4.RELEASE</spring-framework.version>
```

declares a `spring-framework.version` constant with value `3.2.4.RELEASE` and it is later used in dependencies to indicate the version of Spring that this project depends on:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>${spring-framework.version}</version>
</dependency>
```

Dependencies show the list of artifacts that are needed in order for the project to work. The difference between *dependencies* and *dependencyManagement* is that the latter can be also applied to sub-projects. More on it can be read in Maven [manual section about dependency management](#).

But let's get to the most interesting section--**Build**. We've already seen that you don't provide commands that can be run in **pom.xml**, instead Maven decides which goals to provide, based on the packaging method and enabled plugins in this section. Here we have *compiler*, *surefire*, *war*, *eclipse*, *assembly* and *tomcat7* plugins enabled. Each of the plugins provides

additional commands that you now can run on this project.

Typically working with Maven means running a command with some options and goals.

```
mvn [options] [<goal(s)>] [<phase(s)>]
```

It's enough to only remember a small amount of goals to be solid with Maven, thanks to the concept of so called "build lifecycles", which makes the process of building and distributing artifacts clear and simple. Here we will get familiar with the *default* build cycle that handles our project deployment, but there are two other build lifecycles available out of the box: *clean* and *site*. The former is responsible for project cleaning, and the latter for generating the project site's documentation.

BUILDING YOUR APPLICATION WITH MAVEN

The default build cycle consists of 8 phases: *validate*, *compile*, *test*, *package*, *integration-test*, *verify*, *install*, *deploy*. They are pretty self-explanatory, even for non-technical folks, and every phase includes all the previous ones. So if you run,

```
mvn compile
```

in order to compile your project, Maven will first validate and then compile it. All the information about how to compile (against what Java version, where are the source files and so on) is available in **pom.xml** and you don't have to specify it anywhere else. So as you see, you don't have to know anything about the project to be able to achieve some of the goals available out of the box.

Running **unit tests** is also an easy task, as it is included in default lifecycle. Just run `mvn test` and Maven will run all the unit tests wherever they are in the project.

And you may have already guessed that **packaging the application** comes from simply running `mvn package`, which tells Maven to create an artifact based on packaging value provided in `pom.xml`.

By default artifact name will match the pattern: `<artifactId>-<version>.<packaging>` => `spring-petclinic-1.0.0-SNAPSHOT.war`. But in our case it is different: `petclinic.war`, because the *maven-war-plugin*, that is responsible for creating war files artifacts, is given a *warName* parameter, that defines the filename for resulting artifact.

ADDITIONAL MAVEN FEATURES PROVIDED BY PLUGINS

If you check out `readme.md` in the project, you will see that it suggests you to run

```
mvn tomcat7:run
```

to start your Pet Clinic app locally. This goal (`tomcat7`) does not belong to the default build lifecycle, but is provided by a Tomcat plugin. We don't have this goal defined anywhere inside the build script, but rather it's contained inside the plugin itself. To learn what commands are provided by a plugin, you can ask Maven to list them for you:

```
mvn help:describe -Dplugin=org.apache.tomcat.maven:tomcat7-maven-plugin
```

The string after the equals (=) sign is plugin *groupId* and *artifactId* concatenated by a colon (:).

HOW TO SHARE YOUR ARTIFACTS

Once you have your artifact built, you will likely realize that you'd like to share it with the outer world. For development teams in commercial organizations, you can install and [configure your own repository manager](http://maven.apache.org/repository-management.html) (<http://maven.apache.org/repository-management.html>), such as Artifactory or Nexus, i.e. according to your organization's tool kit.

Open source projects have it easier, of course. They can publish to an existing, **public repository** that accepts the uploading of artifacts, such as the [Sonatype OSS Maven Repository](#).

It's up to you to set these options up, and we won't cover the repository software installation here, but as soon as it is ready you will have to add some lines that look something like this example below into **pom.xml** of your project:

```
<distributionManagement>
  <repository>
    <id>deployment</id>
    <name>Internal Releases</name>
    <url>http://yourserver.com:8081/nexus/content/
      repositories/releases</url>
  </repository>
  <snapshotRepository>
    <id>deployment</id>
    <name>Internal Releases</name>
    <url>http://yourserver.com:8081/nexus/content/
      repositories/snapshots</url>
  </snapshotRepository>
</distributionManagement>
```

From here, Maven will upload the built artifact into your repository once you run `mvn deploy` and based on whether your version is a *snapshot* or *release*, it will go to either of the two repositories.

BASIC ASSUMPTIONS AND RESTRICTIONS TO KEEP IN MIND WITH MAVEN

As we see the build script for a demo project we have is already quite big, but you don't have to write it yourself by hand. Nowadays, most of the file gets generated, and the main 3 IDEs can actually insert dependencies more or less automatically as you write import statements in your Java code. Here you can also automatically add plugins into **pom.xml**.

The biggest drawback of Maven continues to be “downloading the internets”, although with some configuration on your side, [Sonatype promises](#) that this doesn't have to be the case. Each dependency and plugin entry in the build script resolves into downloading an artifact and its dependencies (with further dependencies of dependencies). So the first build of the project can take a lot of time and reduce your disk space significantly if you aren't careful.

Maven is also somewhat limited in running custom commands during build steps. You will have to use a plugin that provides this feature, eg. the *antrun plugin*, to do so or write your own plugin. We will cover these cases further in the report.

Going from zero to hero with Gradle

Finding and installing Gradle is quite easy. Like with other build tools, you can just go to the website, download the .zip package, extract it and you're done (i.e. assuming that you have Java installed and working, of course). To make everything more seamless, add your `<gradle install dir>/bin` to the PATH as well.

INSTALLING GRADLE FROM THE WEBSITE

You can download the tool from <http://www.gradle.org/downloads>.

The Gradle website also allows you to install older versions of Gradle.

DOWNLOADS

Current Stable Release

The latest release of Gradle is 1.9, released on 19th November 2013.

Gradle releases come in three flavours:

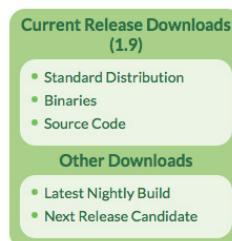
- `gradle-1.9-all.zip` (binaries, sources and documentation)
- `gradle-1.9-bin.zip` (binaries only)
- `gradle-1.9-src.zip` (sources only)

Please read the [release notes](#) before downloading.

Previous Releases

The download links above and to the right are for the latest Gradle release (1.9). To download a previous release, select the version number from the dropdown below.

Choose Version ▾



INSTALLING GRADLE THROUGH CLI

It is also possible to install Gradle using some package manager on Linux or Mac OS, but as with Maven it's possible that these package managers might give you some older version of the tool than what you're looking for.

To install gradle on Ubuntu or Debian, just execute:

```
sudo apt-get install gradle
```

Gradle should be available in repositories for other major Linux distros as well.

INSTALLING GRADLE FROM IDES

Gradle has plugins for Eclipse, IntelliJ and also Netbeans, but the plugin does not come with Gradle itself. You need to download and install Gradle separately.

Gradle's IDE plugin generates the necessary configuration metadata for an IDE, so you can just import the project later and everything is already in place.

USING GRADLE WITHOUT INSTALLING IT

One very cool thing that we discovered about Gradle is the **Gradle Wrapper**. Basically, it is possible for anyone to only check out the project from SCM and start managing it with Gradle, without actually installing Gradle. This is enabled via the Gradle wrapper, a tiny piece of software (about 4kb in size), that can be bundled with the project. In this way, it is also possible to have Gradle basically everywhere the environment is supported. This also means that there is no need to beg your System Administrator to install Gradle on your CI server.

You can get Gradle Wrapper installed when you add these lines to your "build.gradle" file:

```
task wrapper(type: Wrapper) {
    gradleVersion = '1.9'
}
```

WRITING THE BUILD SCRIPT FOR GRADLE

One thing that differentiates Gradle from Maven and Ant is that its build script does not need to have any boilerplate inside. Actually, even an empty text file is a valid Gradle build script! The Gradle build script can be named simply **build.gradle** and it uses Gradle's [own domain specific language \(DSL\)](#) to get the job done.

It's also possible to write plain Groovy inside these scripts, which makes it easy to write custom tasks. However, you need to keep in mind that it establishes what will be from now on a somewhat thin line between manageable and unmanageable scripts. If you come from a 10-year old Ant world, it might take some time to move from XML to this DSL. Fortunately, Gradle's documentation is well-written and contains a lot of code samples.

If you want to have a basic build script for your java project, then the script can just look like this:

```
apply plugin: 'java'

group = 'org.springframework.samples'
version = '1.0.0-SNAPSHOT'

description = "spring-petclinic"
```

BUILDING YOUR APPLICATION WITH GRADLE

To understand how the tool works in real life and on a real project, we again used Spring's Pet Clinic sample application as a project for our case study. We've provided the short steps needed here to build your application.

The first step is clone the project from GitHub repo: <https://github.com/spring-projects/spring-petclinic/>. Then execute the following command:

```
gradle init
```

This feature is in incubation phase still, but it apparently works in most cases. The command is designed to generate a **build.gradle** file based on Maven settings. It's not foolproof, because the generated **build.gradle** file need to try to match the same build logic that you had with Maven.

Now add `apply plugin: 'war'` to your **build.gradle** file, since Pet Clinic is a web application and not automatically recognized. Next is to build the project with Gradle:

```
gradle build
```

To run tests with Gradle, just follow your intuition and run: `gradle test --` Likewise, if you'd like to skip unit tests while building with Gradle then you can just skip the "test" task: `gradle build -x test`

HOW TO SHARE YOUR ARTIFACTS

As Gradle does not yet have its own repositories and repository format (it currently uses Maven Central and its format), then it is possible to either publish artifacts to Ivy or Maven repositories. We'll show you the basics here, but [check out the complete build script](#) as well.

Support for both repositories is achieved using plugins:

```
apply plugin: 'ivy-publish'
```

or

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

And also, it is possible to customize publishing rules:

```
publishing {
    publications {
        maven(MavenPublication) {
            groupId 'org.springframework.samples'
            artifactId 'petclinic_sdk'
            version '1.0.0-SNAPSHOT'
            from components.java
        }
    }
}
```

BASIC ASSUMPTIONS AND RESTRICTIONS TO KEEP IN MIND WITH GRADLE

We wanted to discuss the issue of Tool Migration with respect to Gradle, because we must assume that in many use cases it is a migration story. Developers using Maven or Ant have heard about Gradle and want to try it--after all, we are a curious folk and enjoy testing/breaking new tools. Plus, having learned from all the mistakes and complaints made in the past by Ant and Maven, Gradle has a great opportunity to do it right this time.

As it is almost impossible to figure out automatically what is getting done in Ant build scripts, Gradle does not have any official migration tool for that. But fortunately, there is good old `gradle init` that can read Maven's `pom.xml` and generate a corresponding `build.gradle` file.

One restriction for some developers will be a lack of familiarity with the Groovy programming language. With Groovy in Gradle, it is possible to use existing Ant targets, Maven plugins and even Java classes, without spending too much time and effort on migrating and rewriting everything. Sure, Groovy is powerful and you could write anything you possibly want with it, but at the same time it is possible to write a bunch of spaghetti that nobody understands and no one is able to manage.

Gradle is definitely more programmer-friendly than sysadmin-friendly, because its scripting language is not so verbose and intuitive and requires more understanding about programming and its good practices.

Going from zero to hero with Ant + Ivy

Ant, the mother of the modern build tool for Java, has had its fair share of heat thrown at it over the years, from complaints about the syntax being too rigid to being too verbose and unfriendly for newcomers. While those concerns have their merits, here's a small run through of how to actually get started with Ant; and alongside it the plugin Ivy, which adds dependency management functionality to Ant.

INSTALLING ANT FROM THE WEBSITE

If your IDE of choice doesn't include Ant, you want to use a different version, or you simply prefer building your project from the command line, then you can download it from the Apache website:

<http://ant.apache.org/bindownload.cgi>

Binary Distributions

Apache Ant™

Apache Ant is a Java library and command-line tool that help building software.

Downloading Apache Ant

Use the links below to download a binary distribution of Ant from one of our mirrors. It is good practice to verify the integrity of the distribution files, especially if you are using one of our mirror sites. In order to do this you must use the signatures from our main distribution directory.

Ant is distributed as `zip`, `tar.gz` and `tar.bz2` archives - the contents are the same. Please note that the `tar.*` archives contain file names longer than 100 characters and have been created using GNU tar extensions. Thus they must be untarred with a GNU compatible version of `tar`.

In addition the JPackager project provides RPMs at their own distribution site.

If you do not see the file you need in the links below, please see the master distribution directory or, preferably, its mirror.

Mirror

You are currently using <http://mirror.hosting90.cz/apache/>. If you encounter a problem with this mirror, please select another mirror. If all mirrors are failing, there are backup mirrors (at the end of the mirrors list) that should be available.

Other mirrors:

Current Release of Ant

Currently, Apache Ant 1.9.2 is the best available version, see the release notes.

Note

Ant 1.9.2 was released on 12-July-2013 and may not be available on all mirrors for a few days.

Tar files may require gnu tar to extract

Tar files in the distribution contain long file names, and may require gnu tar to do the extraction.

- `zip` archive: [apache-ant-1.9.2-bin.zip](#) [PGP] [SHA1] [SHA512] [MD5]
- `tar.gz` archive: [apache-ant-1.9.2-bin.tar.gz](#) [PGP] [SHA1] [SHA512] [MD5]
- `tar.bz2` archive: [apache-ant-1.9.2-bin.tar.bz2](#) [PGP] [SHA1] [SHA512] [MD5]

Installing Ant is as straightforward as extracting the downloaded Ant archive to your desired destination (`<ant-install-path>`). After doing so you should have access to run the tool using the `ant` executable located in the `bin` folder; for that reason it could be a good idea to add `<ant-install-path>/bin` to your `PATH`, so you can invoke the tool without having to specify the full path every time.

INSTALLING ANT FROM IDES

If you use an IDE to do all your development and building, then you're most likely in luck, as Ant is included with most IDEs, including Eclipse, NetBeans and IntelliJ IDEA, and have built in support for creating the build scripts.

INSTALLING IVY

If you want to include some dependency management-fu to Ant, you need to install Ivy from the Apache website:

<http://ant.apache.org/ivy/download.cgi>

Afterwards extract the Ivy jar file from the downloaded archive, and place it in the `.ant/lib` folder in your home folder; note you might have to create the folder first! Placing it there should make it available to all Ant installations on your system, including those bundled with IDEs.

Alternatively you can extract it to `<ant-install-path>/lib`, but then it will only work out of the box with that Ant installation.

A NOTE ABOUT ANT'S BUILD SCRIPT

Ant uses XML to define its build script, which by default is named **build.xml**. The script is divided into separate targets that each perform individual actions; common target names include things like clean, build, jar, war, and test. Ant doesn't hand any of these to you automatically, you have to create them all yourself, but it does provide the most common functionality needed to perform the actions.

The basic layout of a **build.xml** file looks something like:

```
<project name="Name" default="build" xmlns:ivy="antlib:org.apache.ivy.ant">
  <!-- Setup of properties/variables used -->
  <property name="classes" location="target/classes" />
  <property name="src" location="src/main/java" />

  <!-- Defining the individual targets -->
  <target name="build" description="...">
    <!-- call javac etc -->
  </target>
  <target name="clean" description="..." />
</project>
```

Here `<project>` is the root tag, containing the name and description of the project, the default target to call, as well as specifying additional XML namespaces, to easily import plugins like Ivy. Also, `<property>` is used to set up immutable properties that are used in the build script, for instance the location of the source code as listed above. And `<target>` is used to define the invokable targets.

WRITING THE BUILD SCRIPT FOR ANT

For creating the build script, we'll continue using Spring Pet Clinic as the example (available on GitHub at <https://github.com/spring-projects/spring-petclinic/>), and show the steps required to create the scripts required to build that project. In order to build it we'll need a few targets: we need to **build** the source code, package it all into a **war** file, be able to **test** the code, and likely want to be able to **clean** up again.

To start with, let's define some **common properties** that we'll need in order for Ant to be able to find our source code etc. PetClinic uses the "Maven way" for source and resource locations, so we'll define the properties accordingly and set the projects default target to create the war file. Likewise, the XML namespace "ivy" is defined, referencing the Ivy plugin, so we can use Ivy tasks in the build script:

```
<project name="PetClinic" default="war" xmlns:ivy="antlib:org.apache.ivy.ant">
  <property name="src" location="src/main/java" />
  <property name="resources" location="src/main/resources" />
  <property name="webapp" location="src/main/webapp" />
  <property name="test.src" location="src/test/java" />

  <property name="target" location="target" />
  <property name="classes" location="${target}/classes" />
```

Next up, creating the **individual targets** for building it all, starting with the build, which invokes the `<ivy:classpath>` task to get a reference to the required dependencies, so they can be used on the class path for the `<javac>` task. You'll also notice that there are references to the properties defined above using the `${property.name}` syntax:

```
<target name="build" description="Compile everything">
  <mkdir dir="${classes}" />
  <ivy:classpath pathid="ivy.path" />
  <javac srcdir="${src}" destdir="${classes}"
        classpathref="ivy.path" includeantruntime="false" />
</target>
```

For **creating the war file**, we utilize the Ant task called `<war>`, which is there to do exactly that: create a war file.

We specify the desired destination with the `destfile` attribute, the location of our `web.xml` file, and then specify what we actually want to include in the war file. The `<classes>` tag is used to specify what goes into the `WEB-INF/classes` folder inside the war file, which in this case is our compiled classes as well as their resources.

Furthermore we use the `<fileset>` tag to indicate that we want to copy everything in the dir specified to the root of the war file. Lastly, we want to copy all the required dependencies to the `WEB-INF/lib` folder, so from Ivy we get the runtime dependencies, and use the `<mappedresources>` tag to add those into the war file, using a mapper that flattens the class path structure, meaning removing all the folder names and only adding the actual files.

The `<war>` task also supports the `<lib>` tag, but since it doesn't support flattening the class path, it is not an optimal way to include dependencies resolved by Ivy.

```
<target name="war" depends="build" description="Create war file">
  <ivy:classpath pathid="ivy.runtime.path" conf="runtime" />
  <war destfile="${target}/petclinic.war"
      webxml="${webapp}/WEB-INF/web.xml">
    <classes dir="${classes}" />
    <classes dir="${resources}" />
    <fileset dir="${webapp}" />

    <mappedresources>
      <restrict>
        <path refid="ivy.runtime.path"/>
        <type type="file"/>
      </restrict>
      <chainedmapper>
        <flattenmapper/>
        <globmapper from="*" to="WEB-INF/lib/*"/>
      </chainedmapper>
    </mappedresources>
  </war>
</target>
```

Since we've set up everything to be compiled to the target folder, **cleaning the project** is as simple as deleting that folder, making the clean target very simple:

```
<target name="clean" description="Delete target">
  <delete dir="${target}"/>
</target>
```


For the **test target**, it's similar to how the build target was set up: we get the dependencies from Ivy, this time requesting the test and provided dependencies, build the tests classes, and then use the `<junit>` task provided by Ant to run the tests; adding the dependencies, resources, compiled classes and test classes to the class path, and tell junit, using `<batchtest>`, to iterate over all classes containing "Test" as part of their name, and run those:

```
<target name="test" depends="build" description="Run JUnit tests">
  <ivy:classpath pathid="ivy.test.path" conf="test,provided" />
  <mkdir dir="${test.report}" />
  <mkdir dir="${test.classes}" />

  <javac srcdir="${test.src}" destdir="${test.classes}"
    classpathref="ivy.test.path" classpath="${classes}"
    includeantruntime="false" />

  <junit printsummary="yes" haltonfailure="no">
    <classpath refid="ivy.test.path"/>
    <classpath path="${resources}" />
    <classpath path="${test.src}" />
    <classpath path="${classes}" />
    <classpath path="${test.classes}" />

    <formatter type="plain"/>

    <batchtest fork="yes" todir="${test.report}">
      <fileset dir="${test.src}">
        <include name="**/*Test*.java"/>
      </fileset>
    </batchtest>
  </junit>
</target>
```

SOME BASICS ABOUT IVY XML FILES

Similar to how Ant uses an XML file for its build script, Ivy also uses XML for its files, which is by default named **ivy.xml**. The minimum Ivy file simply contains an info block with the name of the project. Other information in the Ivy file can be a list of dependencies, information how dependency conflicts are handled, how to artifacts are created from the build, and a list of configurations used to define dependencies' scope.

With this information in mind, a simple Ivy file with a single dependency to Google Guava could look like this:

```
<ivy-module version="2.0">
  <info organisation="myorg" module="mymodule" />
  <dependencies>
    <dependency org="com.google.guava" name="guava" rev="15.0" />
  </dependencies>
</ivy-module>
```

CREATING THE PETCLINIC IVY.XML FILE

In the beginning of the file, we define the **general information** about the project:

```
<ivy-module version="2.0">
  <info organisation="org.springframework.samples"
    module="spring-petclinic" revision="1.0.0-SNAPSHOT" />
```

Since we have a situation with the PetClinic that our dependencies have different scopes; some are only required when running tests, some are only required while compiling, some are provided by the application container at runtime etc. Due to this we have to set up a few different configurations.

Simplified, configurations basically dictate which dependencies are used under which circumstances, and can be used to mimic equivalent functionality of scopes in Maven, by defining the configurations: `default`, `compile`, `provided`, `runtime`, `test`:

```
<configurations>
  <conf name="compile" />
  <conf name="provided" />
  <conf name="runtime" extends="compile" />
  <conf name="test" extends="runtime" />
  <conf name="default" extends="runtime" />
</configurations>
```

With the configurations in place, it's now just a matter of **defining dependencies** and adding those of PetClinic to the Ivy file, specifying which configuration they belong to. Here's an excerpt showing dependencies for some of the configurations:

```
<dependencies>
  <!-- Test dependencies -->
  <dependency org="junit" name="junit" rev="4.11"
    conf="test->default(*)" />
  <!-- ... -->
  <!-- Compile dependencies -->
  <dependency org="javax.servlet" name="jstl" rev="1.2"
    conf="compile->default(*)" />
  <!-- ... -->
  <!-- Provided dependencies -->
  <dependency org="javax.servlet" name="servlet-api" rev="2.5"
    conf="provided->default(*)" />
  <!-- ... -->
</dependencies>
```

The complete file is available here:

<https://gist.github.com/anonymous/5659c066fc5dd71a0ef6>

RUNNING ANT

With the build script and Ivy file created, we can now run Ant to build the PetClinic. Since we specified in the build script the default target to be “war”, invoking that target is as simple as running Ant in the root folder of our project (where we’ve added our **build.xml** and **ivy.xml** files):

```
> ant
```

Similarly, if we want to invoke a specific target, like for instance run the junit tests, which had the target “test” in our script, we just add that target to the command line, like:

```
> ant test
```

ACCESSING ARTIFACT REPOSITORIES

By default Ivy is configured to be able to pull from its local cache as well as the Maven Central, but sometimes you want to be able to pull from other repositories; be that public or private. In Ivy-terms, these are called **resolvers**. To add a resolver you have to create your own **ivysettings.xml** file, and add the resolver to that. If **ivysettings.xml** is placed in the same folder as **ivy.xml**, the settings will automatically be picked up.

Alternatively you can add a `<ivy:settings file="ivysettings.xml" />` task in the Ant build script. Using this method, you can have the settings file located anywhere, thus sharing it across multiple projects, or even have it located remotely and reference it instead via an URL, like `<ivy:settings url="http://myserver.com/ivysettings.xml" />`.

For example, in order to access the NetBeans repository, we need to create an **ivysettings.xml** file and add the repository location (<http://bits.netbeans.org/nexus/content/groups/netbeans/>) to it:

```
<ivysettings>
  <settings defaultResolver="chain"/>
  <resolvers>
    <chain name="chain">
      <ibiblio name="central" m2compatible="true" />
      <ibiblio name="netbeans" m2compatible="true"
        root="http://bits.netbeans.org/nexus/content/groups/netbeans/" />
    </chain>
  </resolvers>
</ivysettings>
```

Ivy supports a plethora of resolvers besides `<ibiblio>`, but it's one of the most commonly used as it supports Maven repositories, and knows how to parse pom files for transitive dependencies. Some of the more trivial resolvers are `<url>`, `<filesystem>`, and `<sftp>`, which, in conjunction with a pattern, can locate the artifact in the referenced location. Of non-trivial resolvers can be mentioned the relatively newly added `<updatesite>` and `<obr>`, which are used, respectively, to fetch dependencies from Eclipse Update sites and OSGi Bundle Repositories.

BASIC ASSUMPTIONS & RESTRICTIONS TO KEEP IN MIND

Out of the gate, it might seem like a daunting task to have to write entire build scripts and Ivy files just in order to get a simple build going. And while Ant doesn't really give you anything for free out of the box, meaning you have to piece most of it together yourself, in the end you are left with a build tool where you're more or less in complete control how things are done, and in what order. You are not forced to place your source code in specific folder, nor adhere to strict folder layouts, but on the other hand, you have to specifically tell the build tool where everything is located.

While the build scripts can seem somewhat restrictive at times, a vast array of plugins does open up a lot of possibilities, from simple stuff as adding for-loop like structures to the script, to deploying war-files to application servers or do dependency management! Basically everything is possible; and if it's not readily available as a core feature or as a plugin, and you're brave enough, you can always write your own plugin, so you can finally get it to automatically start the coffee machine when you start the build.

CHAPTER III:

GETTING YOUR HANDS DIRTY WITH PLUGINS, EXTENSIONS AND CUSTOMIZATIONS

Some of you have skipped Chapter II and landed here, where we go a bit deeper into the discussion about which **plugins are good** to consider for each tool, how the **documentation & community** for each tool is managed, and, for added measure, how to extend your build tool even further by **creating your own plugins...**

Additional plugins of interest for Maven, Gradle and Ant

WHAT TO KNOW ABOUT PLUGINS FOR MAVEN

Maven has a lot of plugins, which are listed on <http://maven.apache.org/plugins/> including those created outside the Maven Project by some third parties. Still this is not the full list. Because of this, Maven is sometimes referred as “plugin execution framework”.

Some interesting plugins include:

- **antrun** - allows to run Ant targets as a step of a build.
- **dependency** - allows manipulating dependencies, such as copying, unpacking and analysis.
- **Eclipse** - can generate Eclipse project file for current project.
- **patch** - can apply patches to source code using GNU patch tool.
- **Apache Tomcat** - allows starting tomcat with your application deployed.

These are not the most used or important ones. It's just interesting to know that you can even apply patches through Maven. You don't have to download anything in order to install a plugin. Just add it into **pom.xml** as it was done in Pet Clinic with Tomcat or any other.

WHAT TO KNOW ABOUT PLUGINS FOR GRADLE

Gradle's key plugins are listed on the website: <http://www.gradle.org/>. Some plugins we like are:

- **Java Plugin** - compiling, testing and publishing Java based software;
- **Eclipse Plugin** - integrating with the Eclipse IDE for development;
- **Maven Plugin** - publishing to Maven dependency repositories;
- **Project Reports Plugin** - generating reports containing useful information about your build.

But, there are other cool plugins too, developed and published by the community. The list can be found here: <http://wiki.gradle.org/display/GRADLE/Plugins>.

We took a quick look at these plugins we would like to highlight five of them:

- **Gradle Git plugin** - helps to perform actions around Git repositories (tagging, versioning);
- **Fingerprint plugin** - adds a file's checksum to its name and also updates references to it. Useful when developing web apps with Javascript;
- **Duplicates plugin** - locates duplicate libraries in dependency tree. Helps to maintain and keep it clean;
- **Jenkins plugin** - adds a possibility to configure Jenkins jobs via Gradle. Also allows to use templates for repetitive operations.
- **Templates plugin** - contains lots of template tasks for creating a fresh project.

WHAT TO KNOW ABOUT PLUGINS FOR ANT

As we already mentioned, Ant enjoys the benefit of having been around for over 10 years now, and has over the time gathered a nice array of plugins; some under the Apache umbrella – either as separate projects or as part of the Ant Libraries project – and others as external tools and tasks. On their website they display a list of some of the external plugins, located at <http://ant.apache.org/external.html>. Other external plugins exist besides what's available on that list though; your favorite search engine most likely knows where.

Which plugins are most useful to you, will most likely vary from project to project; you might not need any plugins at all, and chances are you won't need a plugin to help you deploy to an application server if you're only doing desktop applications. Yet here are some plugins that do bear mentioning and that you might find usable (the plugins are all over the place):

- **Ivy itself!** – while Ivy has already been covered earlier, it still bears mentioning as it is not a standard part of Ant, and provides an often needed dependency management system to the Ant build tool. Ivy can be found at <http://ant.apache.org/ivy/>
- **Ant-Contrib** – While the Ant-Contrib plugin might not have been worked on for a while, it's still a plugin used in many places, especially for tasks providing logic control mechanisms, like for and foreach-loops. Ant-Contrib can be found at <http://ant-contrib.sourceforge.net/>
- **Catalina Ant, jetty-ant, etc** – Ant plugins for controlling and deploying to application servers, for instance for Tomcat (<http://wiki.apache.org/tomcat/AntDeploy>), or Jetty (<http://www.eclipse.org/jetty/documentation/current/jetty-ant.html>)

- **ProGuard** – Ant plugin for shrinking, shading and obfuscating your code, available from <http://proguard.sourceforge.net/>
- **Apache Compress Antlib** – Ant plugin that enables you to compress or uncompress files, besides the standard zip. Uses the Apache Commons Compress library, and thus supports the same compression types, including gzip, bzip2, and 7z. Available at <http://ant.apache.org/antlibs/compress/index.html>

Community, documentation and materials for your build tool

DOCUMENTATION AND COMMUNITY WITH MAVEN

We have seen many times people being not pleased with Maven documentation, using words like poor, rough, outdated, lacking examples and details when speaking of it. Our opinion is the documentation is decent. It can be overly scientific sometimes, but as a beginner you can get a nice quick introduction to the topic.

We can also recommend check out [Sonatype's Maven reference](#). If docs are not good enough somewhere, then there are already a lot of good how-tos, blog posts and books on topic. Since Maven has been around for nearly a decade, there are some people that feel very comfortable with Maven and can answer your questions, if you ask them on QA sites or forums.

DOCUMENTATION AND COMMUNITY WITH GRADLE

We don't want to be overwhelmingly positive here about Gradle but to be honest, Gradle's documentation is pure awesomeness. It is actively maintained and very well-formatted. It can be downloaded as a single PDF or HTML file, so it is possible to easily browse it while being offline and free of distractions that internet tends to offer us all the time. Everything needed to find documentation, support and other information can be found from the website.

Gradle is a new "hot thing" in the world of build tools and looks like it will get more attention and usage in the near future. The good example is the Google's decision to move building of Android to Gradle, which was announced at [Google I/O 2013 - The New Android SDK Build System](#).

Because Gradle development and adoption is the primary focus of a company called [Gradleware](#), which offers commercial support and

consultancy services, our experience with documentation is good, with everything is up to date, maintained and corresponds to the current version of the tool. Also, trainings are taking place around the world, organized by Gradleware. You can take a look at "Upcoming Trainings" section on the website.

DOCUMENTATION AND COMMUNITY WITH ANT + IVY

Overall, the manual for Ant is very good (available online at <http://ant.apache.org/manual/> or downloadable from <http://ant.apache.org/manualdownload.cgi>); there is full documentation for all the individual tasks, including mostly useful example on how to use them. While the overall look of the manual doesn't exactly scream contemporary – it has a very JavaDoc style feel to it – it's functional and provides the necessary information without much fuss.

Unfortunately, the same cannot be said for the Ivy documentation (available online at <http://ant.apache.org/ivy/history/latest-milestone/index.html>). While all the individual tags are listed in the documentation, the quality of the information varies greatly from tag to tag; some only contain a single one-liner description and a list of possible attributes, while other have comprehensive descriptions, with links to related topics, and several examples of usage. Furthermore, the documentation is often littered with notes like "Since 1.4". While interesting that this specific sub-feature has been available since that version, it doesn't really justify being that heavily emphasized, especially when looking at the documentation for version 2.3. The being said, navigation is easy, so finding the page with information is fast; if you know what you're looking for!

How to write your own custom logic and plugins

CUSTOM PLUGIN DEVELOPMENT WITH MAVEN

As we already know, Maven does not support any custom goals/targets as Ant or Gradle. You can't add a new goal directly into **pom.xml** and set a list of commands that correspond to it. The question arises, what we should do, if we need some custom goal and no-one has previously written a plugin for that. There are a couple of options.

Maven has the **Antrun** plugin, which allows you to run Ant tasks from Maven. You can write your custom logic directly into *pom* or, what is better, create a **build.xml** file similar to ordinary Ant build script and call targets from *pom* using `<ant>` tag. This is useful when you have a small amount of custom tasks to add and they are actually not so connected to each other.

When the number of custom tasks rises, Ant scripts and calls start to pollute **pom.xml** and a better way should be extracting the logic into a plugin. It is also more comfortable for some developers to write a plugin in Java (or Groovy), than some scripts in XML. Maven has a [good introduction into plugin development](#) as well as a part in the [Sonatype Maven reference](#).

As a quick introduction on **how to write your plugin**, we will generate a default "Hello World" plugin, and go through a bit of the code to fully understand it. Maven has a special archetype for projects that later will be packaged as Maven plugins. You should generate such a project by running:

```
mvn archetype:generate \
    -DgroupId=sample.plugin \
    -DartifactId=hello-maven-plugin \
    -DarchetypeGroupId=org.apache.maven.archetypes \
    -DarchetypeArtifactId=maven-archetype-plugin
```

You can change *groupId* and *artifactId* as you like. Now you should have *hello-maven-plugin* directory created - this is the project root. You will also find **MyMojo** class with **execute** method in `src/main/java/sample/plugin/` directory inside the project (e.g. it should look like this <https://gist.github.com/DracoAter/7661033>). This execute method is an entry point for your plugin. The plugin does nothing but touches a **touch.txt** file in project build directory.

Maven uses **@annotations** to define some special attributes, such as:

```
@Parameter( defaultValue = "${project.build.directory}", property
= "outputDir", required = true )
private File outputDirectory;
```

The **@Parameter** annotation identifies an attribute, so that when you use the plugin you can set your own output directory. Another annotation used here is **@Mojo** and it defines a goal to run in order to use the plugin.

Before you can **use your plugin** it should be built and put into some repository where Maven can download it. As we will be using it locally, we can add it into local repository which is most of the time resides in `$HOME/.m2/`. From here it's just **mvn install**.

Now we can add the following lines into **pom.xml** of some project on your machine (even *hello-maven-plugin*) to enable the plugin.

```
<plugin>
  <groupId>sample.plugin</groupId>
  <artifactId>hello-maven-plugin</artifactId>
  <configuration>
    <outputDirectory>mydir</outputDirectory>
  </configuration>
</plugin>
```

After that, if you run `mvn sample.plugin:hello-maven-plugin:touch` the plugin will be invoked and `touch.txt` file will be created. If you didn't provide `configuration.outputDirectory`, the file will be created in default location as specified by `defaultValue` in the `@Parameter` annotation.

Now, to distribute your awesome new plugin, just share it like you would any other artifact.

CUSTOM PLUGIN DEVELOPMENT WITH GRADLE

Gradle is a very general-purpose build tool that can be basically used for automating everything. Although it looks more like another “java build tool”, it actually isn't only for Java. Building Java projects and its dependencies is also implemented as a plugin.

Writing plugins for Gradle is really enjoyable and you can quickly test them out in place without a need to set up some Java project for it first. Just write in in place, test it out and then refactor it out wherever you want.

To implement a plugin in Gradle, you need to write an implementation for the “Plugin” interface:

```
class HelloPlugin implements Plugin<Project> {
  void apply(Project project) {
    project.task('hello') << {
      println "Hello World!"
    }
  }
}
```

And then, just apply the plugin to your project with:

```
apply plugin: HelloPlugin
```

When writing Gradle plugins, there are three options where to put the source code and how to package the plugin:

1. Include the source in the build script (makes easy to refactor build scripts later);
2. Create a folder (buildSrc) for plugin's source code.
Gradle will discover the code, compile and run the plugin;
3. Build a plugin as a separate project.

Also, plugins can be written in Groovy, Java or Scala or another language that will end up as Java bytecode.

CUSTOM PLUGIN DEVELOPMENT WITH ANT + IVY

Ant has been around for quite a while, and has seen its fair share of development, not only on the product itself, but also 3rd-party contributed plugins. Ivy itself started as a 3rd-party plugin, but was, for its 2.0 version, pulled into the Apache Ant fold.

That said, there's a good chance that most of your needs are already covered by Ant itself, or by one or more of the publically available plugins out there; but should that not be the case, you can always pick up the task of writing your own plugin, and integrate it with your build scripts.

It is worth to quickly mention the ant-task `<exec>` here, since that can be used to execute external programs and scripts, which might be able to the custom logic you're looking for -- no need to reinvent the wheel, if you already have a working solution for the problem.

If you choose to go the way of creating a new task for Ant, it is actually fairly straightforward. The easiest way is to create a Java Class with a null-args constructor and a `public void execute()` method. Afterwards, you can use the `<taskdef>` task in Ant to reference that class, and thus define your custom task.

So writing a simple Hello World task:

```
public class HelloWorld {
    public void execute() {
        System.out.println("Hello World");
    }
}
```

And to use it, from a target in Ant:

```
<target name="hello">
    <!-- define the task call helloworld -->
    <taskdef name="helloworld" classname="HelloWorld"
            classpath="path/to/helloworld.jar"/>

    <!-- Invoke the newly defined helloworld task -->
    <helloworld/>
</target>
```

Yes, it really is that simple!

While the above code is valid and works, you'd most likely want your task to extend from `org.apache.tools.ant.Task` (or one of its subclasses), by doing so you get access to convenient methods, like being able to log output, getting properties from the project etc.

If your task requires attributes, like if you want to specify which world we're referring to like `<helloworld world="Mars" />`, it's a simple matter of adding a setter methods to the class, as if the attribute was a Bean property. So in this case, adding a `public void setWorld(String world)` method to the class would enable you to set the world attribute.

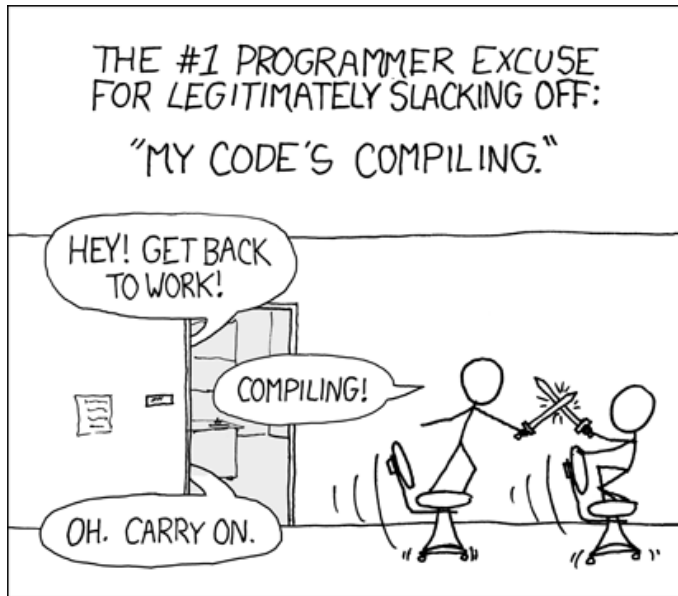
This is the very basic requirements for creating your own task in Ant. Of course you can create far more complex tasks than the above, this is simply the first small step to take to get started with it; only the sky is the limit...your imagination, and potential lack of ability to implement said imagination, might also add some limiting factors :)

CHAPTER IV:

SUMMARY, CONCLUSION AND GOODBYE COMIC :-)

For you lazy/busy coders, here is your TL;DR section--complete summary of each chapter, along with a conclusion which outlines the best (and worst) parts that we saw about each tool. Also, we'll give you a brief preview of **Java Build Tools - Part 2**, which will include a full feature comparison, performance testing and ranking for decision makers in early 2014.

Summary of what we've seen so far



Source: <http://xkcd.com/303/>

CHAPTER I - AN INTRODUCTION AND [RATHER] SHORT HISTORY OF BUILD TOOLS

Here we introduce the concept of a build tool, what they are supposed to do for us as developers and, as app complexity has increased over time, why dependency management quickly became a high priority issue to handle correct.

We discuss a bit on the DSL (Gradle) vs. XML (Maven, Ant) debate and give a timeline on the evolution of build tools since the late 1970s. Since ZeroTurnaround has been looking into the use [and misuse] of different Java tools and technologies, we show our Build Tools landscape findings from surveys completed in late-2010, early 2012 and mid-2013.

To finish up, we give a quick overview of Maven, Gradle and Ant + Ivy, the three build tools this report covers (with a small shout out to SBT as well).

CHAPTER II - HOW TO GET SET UP WITH MAVEN, GRADLE AND ANT + IVY

Here we show you how to go from zero to hero with each tool, from installation via website, CLI and IDE to writing the build script for each tool using the well known Spring Pet Clinic sample application, to actually building your application.

Using repositories, we show you how to share your artifacts and cover some basic assumptions and restrictions to keep in mind with your choice of build tool.




CHAPTER III - GETTING YOUR HANDS DIRTY WITH PLUGINS, EXTENSIONS AND CUSTOMIZATIONS

For existing users and developers that want to get a bit deeper, we provide a list of the most popular and our favorite plugins to use for each tool. Since accurate, easy-to-access documentation and a useful community is always nice to have with whatever technology you're working with, we review the situation for each tool.

Next, we move on to the customizations for your build tool of choice. We show you how to get started with writing your very own plugins for each tool, and how to get people using them.

Conclusion: the Good, the Bad and the Ugly

After spending this much time with Maven, Gradle and Ant + Ivy, we've come to know each tool relatively well--of course, this doesn't make us anything close to experts on any of these tools, but hopefully we've provided enough of a how-to guide for beginners to follow. So in conclusion, we wanted to finish up with a simple "3 good things, 3 bad things" list for each tool.

	WHAT ROCKS	WHAT LACKS
	<ul style="list-style-type: none">• Maven Central was a major game changer and is still the de facto repository for plugins and anything else you want.• Common project structure and build cycles makes you familiar with all Maven projects if you're familiar with one.• Easy artifact sharing lets you upload to an existing service or manage your own repository (i.e. Artifactory or Nexus).	<ul style="list-style-type: none">• Little customization available, relies on other tools for that.• "Downloading the internets" at risk if you don't read the special documentation prepared because so many people complained• Long pom.xml-s aren't easily readable, especially with multi-module projects.
	<ul style="list-style-type: none">• Lets you ditch duplicated folder structures for multiple-build projects, and enables clean and nice configurations for custom builds.• Gets rid of projects with large build scripts, while still being relatively readable without the boilerplate.• DSL + Groovy enable to write custom and complex logic sometimes needed in Android build scripts.	<ul style="list-style-type: none">• Requires knowledge of Groovy to be able to work with Gradle, presenting a learning curve.• DSL means that a random developer will have a lot of trouble understanding what's going on with your build• There is no Gradle repository or repo format yet, so familiarity with either Ivy or Maven is still needed when doing coding in Java.
	<ul style="list-style-type: none">• You are in complete control of the build process, you decide everything...• No inherent restrictions imposed with regards to directory structure• Customizable in a way that Maven is not, depending on your comfort with plugins	<ul style="list-style-type: none">• You are in complete control of the build process, nothing is handed to you...• Build scripts can easily become long and verbose (try over 130 LoC just to start Ant and Ivy!)• No build script is the same; checking out and building unknown software using Ant can require you to look through the defined targets, to figure out what to actually call in order to make everything run.

Next up: Ranking Build Tools for Features and Use Cases

In **Java Build Tools - Part 2** (to be released in early 2014), we'll run tests and a full feature comparison with both a small and large/multi-module web application. So if the decision on which build tool to migrate to for your next project isn't yet clear, our Part 2 report should help you get there. As a sneak preview, here are the areas that we plan to look at:

1. LEARNING CURVE

When learning a new technology or tool, it's always nice to get up and running really quick making use of features that are both intuitive and easy to use. The only problem is, it's very rare to find them! We'll look at how much knowledge you need to know in advance to use the tools and how easy it is to script something up from scratch.

2. BUILD SPEED

Rome wasn't built in a day, but your projects sure should! The speed of your build which could include a number of tasks is important and often hampered by such pastimes as 'downloading the internetz'. We'll time our builds for our two sample projects and see who comes out on top.

3. BUILD SCRIPT COMPLEXITY

There are many aspects that can add to the complexity of a build script, particularly from the point of view of another person in your team, new to the script you're writing. How easy is it for them to read/understand your script? How many lines of script must they sieve through before finding what they're after?, is the build script flow intuitive or easy to follow?

4. CONTINUOUS INTEGRATION

Anyone who's anyone uses some form of CI server in their build environment, so it's important... wait what? You don't? Are you joking? Anyway, it's really important to get a really good integration between your build tool and your CI server, whichever one you choose.

5. PLUGINS

This section is really a two part section. Firstly how many plugins exist out there for you to plagiarise^H^H^H^H^H^H^H^H^H reuse? It's important you're not just reinventing the wheel for every project, so what have others already done? When a plugin you need doesn't exist, how easy is it to create one from scratch?

6. COMMUNITY AND DOCUMENTATION

A very important part of any tool's adoption is the documentation, as it tells you how you should go about reaching your goals with a tool. The community is just as important as it's built up of people who have walked the same path as the one you're on, and will give you practical advice and experience when the docs are wrong, inaccurate or simply do not apply to your environment!

7. APPLICATION SERVERS INTEGRATION AT DEVELOPMENT TIME

Many projects produce applications that are deployed into a container at runtime. Build tools offer integration, via plugins, to some/many of the major application servers. This section will dig deeper into the breadth of application server support and the richness of what is offered by each build tool.

As with our other reports, we'll rank each build tool in each of the categories above and give scores. Once we have our scores for all categories, we'll compare the relative importance of each category for various types of applications, and try to determine which is the best build tool for use for different projects. Oh, and if you disagree with our category choices or want to see another category we missed, let us know quickly at labs@zeroturnaround.com or [@RebelLabs](https://twitter.com/RebelLabs) and we'll consider adding it into the second report.

THIS REPORT IS SPONSORED BY **JREBEL**

BETTER THAN RAINBOWS IN YOUR PANTS



yeah we said it!

START YOUR FREE TRIAL

NO CREDIT CARD REQUIRED

JRebel

TAKES LESS THAN 1 MIN.



↙ Contact Us

Twitter: @RebelLabs

Web: <http://zeroturnaround.com/rebellabs>

Email: labs@zeroturnaround.com

Estonia

Ülikooli 2, 4th floor
Tartu, Estonia, 51003
Phone: +372 653 6099

USA

399 Boylston Street,
Suite 300, Boston,
MA, USA, 02116
Phone: 1(857)277-1199

Czech Republic

Osadní 35 - Building B
Prague, Czech Republic 170 00
Phone: +372 740 4533

This report is brought to you by:

Juri Timošin, Sigmar Muuga, Michael
Rasmussen, Simon Maple, Oliver White,
Ladislava Bohacova