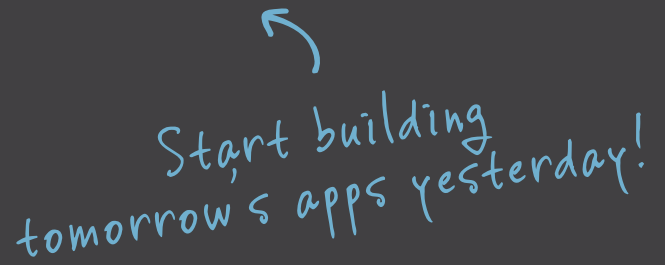
# JAVA BUILD TOOLS: PART 1

## AN INTRODUCTORY GUIDE TO GETTING STARTED WITH MAVEN, GRADLE AND ANT + IVY




REBELLABS

# TABLE OF CONTENTS

# 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 an 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 and providing the developer 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

## **Managing dependencies seems pretty important.** Why is that?

One big area or pain which build tools have taken away/eased is dependency management. What did we do before that 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 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 (Ant's build.xml 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**

1977 - MAKE, the world's first build tool, released

1996 - JDK 1.0 launched

1999 - Apache Software Foundation created & first release of Tomcat

2000 - Ant is born

2002 - Maven 1.0 is launched

2003 - Scala 1.0 is released

2004 - Ivy is introduced

2005 - Maven 2.0 is released

2008 - Simple Build Tool (SBT) first version published

2010 - Maven 3.0 introduced

2012 - Gradle 1.0 launched

Add a did-you-know note to Ant section: 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.

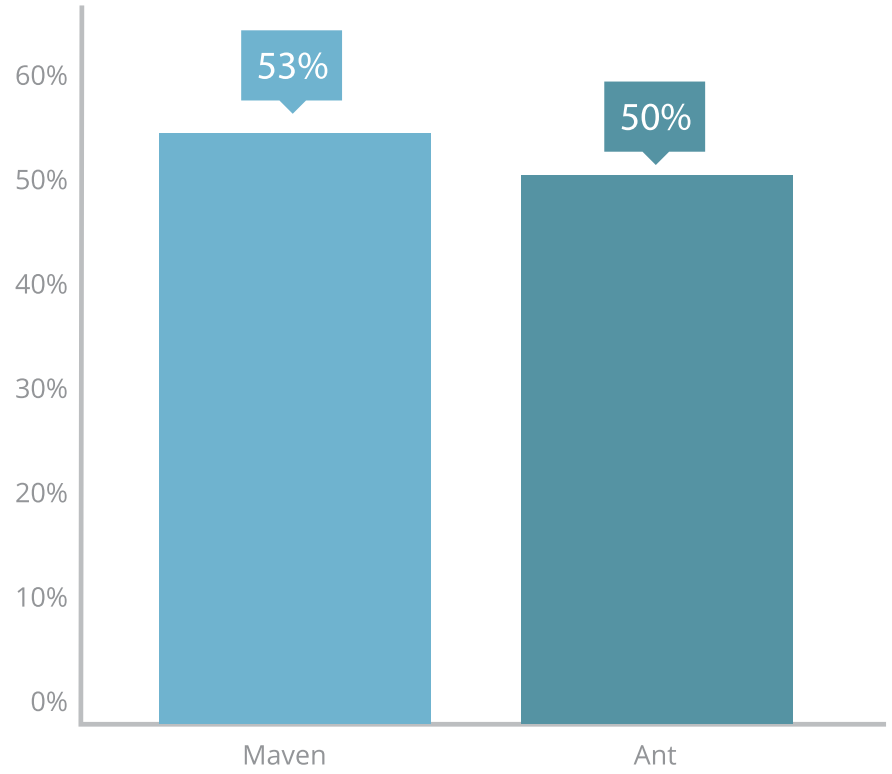
Add a did-you-know note to Maven section: Did you know that Maven was originally written to be a build tool for Apache Turbine web framework? Turbine is long forgotten, but Maven has become the de-facto standard for dependency management in Java.

# **Build tools we'll look at in this report** (and the tools we won't)

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.

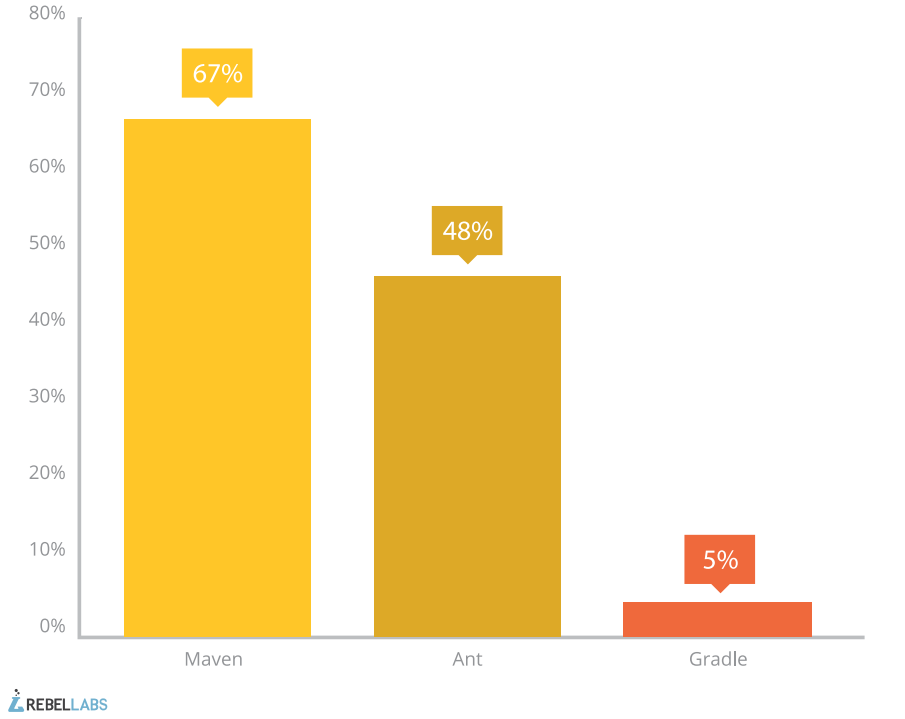
One cool thing is that we've been looking at Build Tools in use since 2010, and we've seen the following trends over the last few years. Here are the self-reported statistics from 3 years worth of developer data.

## Build Tools Popularity - Late 2010



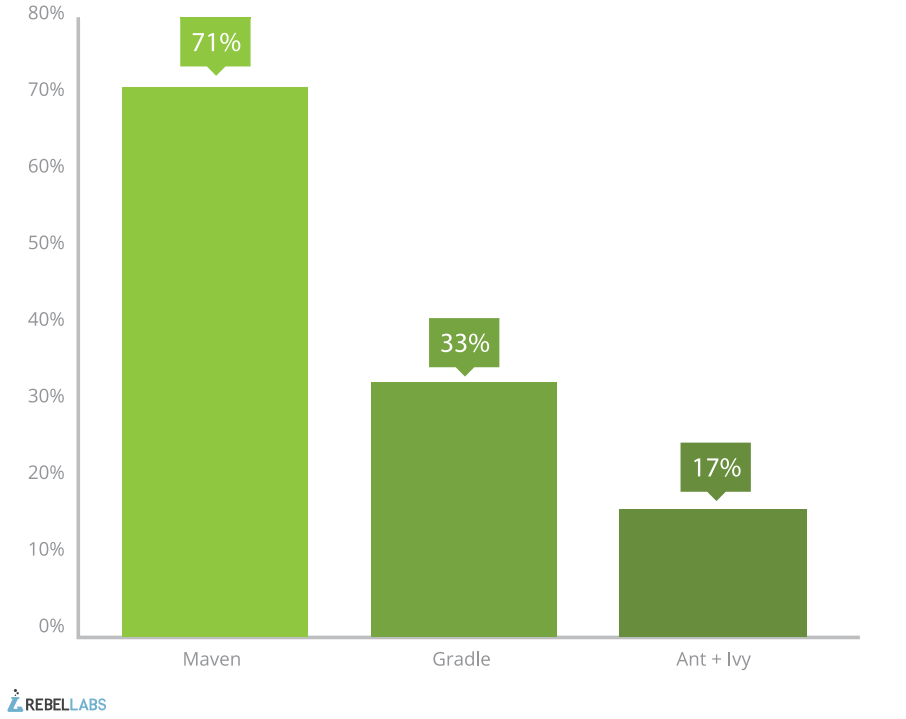

**Sample size:** 1000+ developers
**Source:** Java EE Productivity Report 2011 © ZeroTurnaround
http://zeroturnaround.com/rebellabs/java-ee-productivity-report-2011/

## 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.cz/2013/08/java-build-tools-survey-results.html

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

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, possibly because it is the most used build tool around at the moment among Java developers. Originally, Maven (1) was created in 2002 but really struck a chord with Java developers in 2005 when Maven 2 was released.


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...

## **Special callout to** Simple Build Tool (SBT)

We felt that SBT, with it's 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 the first glance one could think that as it's taken the DSL approach, it's the same as Gradle. The important difference lies in static typing. The build management 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 a adventurous mood, take it to 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.

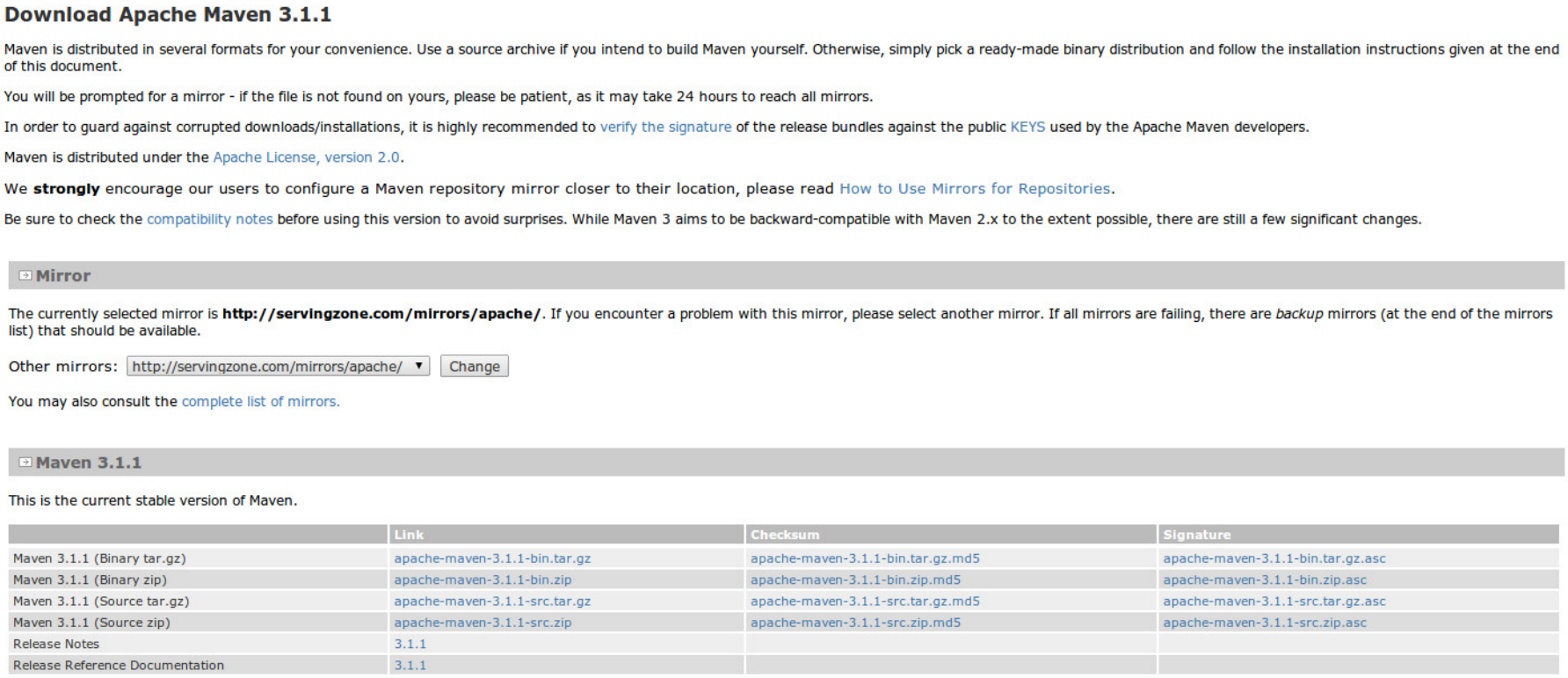
# **Going from zero to hero** with Maven

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.

## 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.

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.



**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: http://servingzone.com/mirrors/apache/ Change

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 | | |

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 itself 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 important to remember only a small amount of goals to be solid with Maven, thanks to the Maven concept of so called "build lifecycles", which make 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.

## 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 ofte have older versions 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 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/SpringSource/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.

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:**
Simon Maple, Adam Koblentz, Sigmar Muuga,
Cas Thomas, Sven Laanela, Oliver White,
Ladislava Bohacova