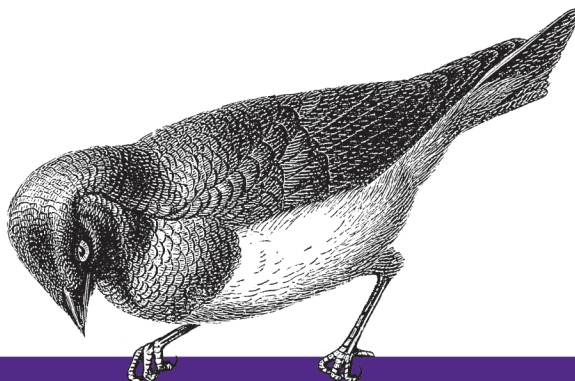
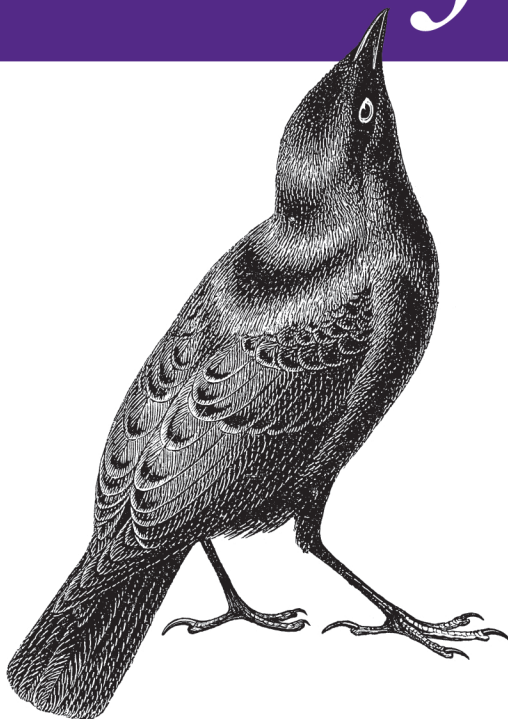


Continuous Integration for Robust Building and Testing

Integrating



PHP Projects with Jenkins



O'REILLY®

Sebastian Bergmann

Integrating PHP Projects with Jenkins

Today's web applications require frequent updates, not just by adding or upgrading features, but by maintaining and improving the software's existing code base as well. This concise book shows PHP developers how to use Jenkins, the popular continuous integration server, to monitor various aspects of software quality throughout a project's lifecycle.

You'll learn how to implement continuous integration to automate processes for building and deploying regular software releases. The book also shows you how to use Jenkins to monitor and improve your application through continuous inspection. You'll come to understand why reducing complexity and eliminating duplicate code is just as important as introducing new functionality.

- Learn how to use Apache Ant to automate your software builds
- Create a job for your PHP project in Jenkins and set up a continuous integration environment
- Add static code analysis tools to your build for continuous inspection
- Use specialized PHP and Jenkins tools to simplify the automated build and continuous integration of your project
- Explore additional processes and techniques, such as adding automated integration tests

Twitter: @oreillymedia
facebook.com/oreilly

US \$19.99

CAN \$20.99

ISBN: 978-1-449-30943-5



5 1 9 9 9

O'REILLY®
oreilly.com

Integrating PHP Projects with Jenkins

Sebastian Bergmann

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Integrating PHP Projects with Jenkins

by Sebastian Bergmann

Copyright © 2011 Sebastian Bergmann. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://my.safaribooksonline.com>). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Editor: Julie Steele

Production Editor: Jasmine Perez

Cover Designer: Karen Montgomery

Interior Designer: David Futato

Illustrator: Robert Romano

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Integrating PHP Projects with Jenkins*, the image of starlings, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-30943-5

[LSI]

1315836072

Table of Contents

Preface	v
1. Build Automation	1
The Example Project	2
Our First Build Script	2
2. Setting Up Jenkins	5
Installing the PHP Quality Assurance Toolchain	5
Installing Jenkins	6
3. Continuous Integration	11
Running Unit Tests During the Build	11
Creating a Jenkins Job	14
4. Continuous Inspection	21
API Documentation	21
Software Metrics	22
Duplicate Code	24
Coding Standard Violations	26
Result Aggregation	29
Complete Build Script	30
5. Automating the Automation	33
PHP Project Wizard	33
Template for Jenkins Jobs for PHP Projects	35
6. Conclusion	37
Continuous Integration and Development Branches	37
Additional Testing	38
Continuous Deployment	40

Bibliography 41

Preface

Why Continuous Integration?

Most web applications are changed and adapted frequently and quickly. Their environment, for example the size and the behaviour of the user base, are constantly changing. What was sufficient yesterday can be insufficient today. Especially in a web environment it is important to monitor and continuously improve the software quality not only when developing, but also when maintaining the software. The practice of Continuous Integration is the solution of choice to achieve this goal.

Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily—leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.

—Martin Fowler

Continuous Integration reduces the amount of repetitive processes the developers need to perform when building or deploying the software thus reducing the risks of late discovery of defects, producing low-quality software, lack of project visibility as well as lack of deployable software.

Why Jenkins?

At least as far as I know, Sebastian Nohn was the first to experiment with the continuous integration of PHP projects. Back in 2006 he [described on his blog](#) how to use [Apache Ant](#) and [CruiseControl](#) to set up a continuous build that invoked [PHPUnit](#) to run the unit tests of a PHP project.

Soon thereafter, the developers of Mayflower GmbH started to build a quality assurance platform for PHP projects that was based on CruiseControl. It not only ran unit tests as part of the build loop but also static code analysis tools such as [PHP_CodeSniffer](#) to collect software metrics over time.

This quality assurance platform gave Manuel Pichler the idea to start the [phpUnderControl](#) open source project. This was a set of patches for CruiseControl (which could

not be customized or extended through plugins) that added out-of-the-box support for PHP projects. `phpUnderControl` significantly lowered the barrier to entry for introducing continuous integration into a PHP project but since it was based on CruiseControl it was a nightmare from an operations point of view and not as customizable and flexible as one would like.

The developers of Mayflower GmbH later refactored their quality assurance platform for PHP projects into a tool that can be used with arbitrary continuous integration servers. It was open-sourced under the name *[PHP_CodeBrowser](#)*.

When the Java community started to migrate from CruiseControl to more modern continuous integration servers, I had a look at Hudson (as *[Jenkins](#)* was called back then) and tried it together with the PHP quality assurance toolchain. I was pleasantly surprised to find a product that was not only more convenient to use and more robust (from an operations perspective) than CruiseControl but to also meet a vibrant development community. Thanks to the plethora of plugins developed and maintained by this community, Jenkins supports building and testing virtually any project, including PHP projects.

The PHP community has developed a wide range of quality assurance tools since 2006. Where Sebastian Nohn only ran PHPUnit tests with Apache Ant and CruiseControl, a typical continuous integration setup for a PHP project today also includes the detection of violations of a defined set of coding guidelines and the gathering of various software metrics as well as the generation of API documentation, for instance.

Over the course of the last two years I have successfully set up continuous integration environments that are based on Jenkins for our customers. These development teams leverage Jenkins to monitor the various aspects of software quality throughout the lifecycle of their projects. This enables them to create, maintain and extend sustainable software of high quality with PHP. The result of this experience is a template for Jenkins jobs for PHP projects that I have released as an open source project.

What's in This Book?

This book covers everything you need to know in order to use Jenkins for the continuous integration of your PHP projects. Here is how the book is organized:

[Chapter 1](#)

In this chapter you will learn how to use Apache Ant to automate your build.

[Chapter 2](#)

This chapter explains how to set up Jenkins and install the PHP tools that are required to continuously integrate a PHP project.

[Chapter 3](#)

This chapter shows how to create a job for a PHP project in Jenkins.

Chapter 4

Building on the continuous integration environment that was set up in the previous chapter, you will learn in this chapter how to add static code analysis tools to the build for continuous inspection.

Chapter 5

This chapter shows how the automated build and continuous integration of a PHP project can be simplified by using the *PHP Project Wizard* and the *Template for Jenkins Jobs for PHP Projects*.

Chapter 6

This chapter concludes the book with a summary of the benefits of Continuous Integration and Continuous Inspection while providing an outlook of what you can implement in addition to the processes and techniques described in this book.

This book makes the assumption that the reader is familiar with the concept of Continuous Integration and the set of problems it aims to solve. [Duvall2007] and [Humble2010] are recommended basic and further reading, respectively, on this topic.

While this book explains how to install Jenkins and how to configure it for PHP jobs it does not cover topics such as authentication, for instance, that are impartial to the programming stack used. Furthermore, the assumption is made that Jenkins is installed in a UNIX environment. For Jenkins-related topics not covered in this book the reader is referred to [Smart2011].

The planning, execution, and automation of tests for the different layers and tiers of a PHP-based web application is also outside the scope of this book. [Bergmann2011] is an excellent resource on these matters.

Finding Out More

If you would like to find out more about Jenkins and PHP, and to get the latest updates, visit this book's companion website at <http://jenkins-php.org/>. This is also where the template for Jenkins jobs for PHP projects is maintained.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

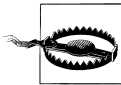
Shows commands or other text that should be typed literally by the user.

Constant width *italic*

Shows text that should be replaced with user-supplied values or by values determined by context.



This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Integrating PHP Projects with Jenkins* by Sebastian Bergmann. Copyright 2011 Sebastian Bergmann, 978-1-449-30943-5.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Safari® Books Online



Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, down-

load chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at <http://my.safaribooksonline.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

<http://oreilly.com/catalog/9781449309435/>

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

Acknowledgments

There are many wonderful people without whom this book would not have been possible: simply because there would have been no tools to write about. These people are Kohsuke Kawaguchi (creator of Jenkins), Greg Sherwood (creator of PHP_CodeSniffer), Manuel Pichler (creator of PHP_Depend and PHPMD), Elger Thiele (creator of PHP_CodeBrowser), and Volker Dusch who co-maintains the template for Jenkins jobs for PHP projects.

Thank you to my technical reviewers. The feedback from Arne Blankerts, Stefan Pribsch, and Volker Dusch was insightful as usual and really improved the book.

Thank you to Julie Steele and everyone at O'Reilly for supporting and encouraging this book.

Build Automation

Continuous Integration requires a fully automated and reproducible build as well as the use of a version control system to be effective. This book makes the assumption that the reader is already familiar with a version control system such as Git.

In this chapter we discuss *build automation*, the practice of automating (scripting) the various tasks that software developers need to perform in their daily routine. These tasks usually include the compilation of source code into binary code and the running of automated tests as well as the packaging and possibly even the deployment of the resulting binaries (PEAR packages or PHAR archives, for instance).

Although PHP is an interpreted language and does not use an explicit compilation step it is common to perform code generation or code transformation tasks during a build nowadays. Scaffolding code generated by a framework or code generated by an object-relational mapping tool as well as autoloader code are common use cases for such generated code.



PHP allows registering a callback which is automatically invoked when a class or interface is about to be used but has not been declared yet. *phpab* is a tool that automatically generates the code for such a callback by analysing the project's sourcecode.

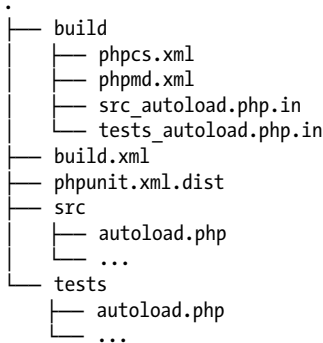
A wide variety of build automation tools exists. I have worked with development teams that were content with using simple shell scripts to automate their builds. I have also seen [GNU make](#), [Apache Ant](#), [Phing](#), [Pake](#), [Rake](#), and custom solutions being successfully used to automate the build of PHP projects. My personal preference is Apache Ant and that is what we will use in the examples in this book.



It poses no problem if you happen to prefer another build automation tool than Apache Ant. You will still be able to use Jenkins to continuously integrate your PHP project.

The Example Project

Throughout this book we will use an example project that is hosted at <http://github.com/thePHPcc/bankaccount>. This is what the project structure looks like:



- The *build* directory contains the XML configuration files for PHP_CodeSniffer (*phpcs.xml*) and PHPMD (*phpmd.xml*) as well as customized templates for *phpab*, the *PHP Autoload Builder*.

PHP_CodeSniffer and PHPMD are static analysis tools for PHP code and are covered in [Chapter 4](#) where we discuss Continuous Inspection.

- *build.xml* is the Apache Ant build script that is used to build the project.
- *phpunit.xml.dist* is the PHPUnit configuration file for the project.



If *phpunit.xml* or *phpunit.xml.dist* (in that order) exist in the current working directory and PHPUnit's *--configuration* switch is not used, the configuration for PHPUnit will be automatically read from that file.

- *src/autoload.php* and *tests/autoload.php* contain the autoloader code for the application and its test suite, respectively.

Our First Build Script

A task that can and should be automated in a build script (because it makes no sense to perform such a task at runtime) is the generation of code for an autoloader. [Example 1-1](#) uses this task to show the essence of writing a build script for use with Ant.

Each project defines one or more *targets*. A target is a set of *tasks* you want to be executed. When starting Ant, you can select which target(s) you want to have executed. When no target is given, the project's default is used. [...] A target can depend on other targets. [...] Ant resolves these dependencies. [...] A task is a piece of code that can be executed.

—<http://ant.apache.org/>

Example 1-1. *build.xml* script that invokes *phpab*

```
<?xml version="1.0" encoding="UTF-8"?>

<project name="BankAccount" default="build">
  <target name="build" depends="phpab"/>

  <target name="phpab" description="Generate autoloader scripts">
    <exec executable="phpab">
      <arg value="--output" />
      <arg path="{basedir}/src/autoload.php" />
      <arg value="--template" />
      <arg path="{basedir}/build/src_autoload.php.in" />
      <arg path="{basedir}/src" />
    </exec>

    <exec executable="phpab">
      <arg value="--output" />
      <arg path="{basedir}/tests/autoload.php" />
      <arg value="--template" />
      <arg path="{basedir}/build/tests_autoload.php.in" />
      <arg path="{basedir}/tests" />
    </exec>
  </target>
</project>
```

In the build script above, we first define a **build** target. This target does not perform any task by itself but rather depends on other targets, so far only **phpab**. You can think of this as a *meta target* that orchestrates other targets.

The **phpab** target uses the `<exec>` task to invoke the aforementioned PHP Autoloader Builder to generate the autoloader code. The two `<exec>` tasks are equivalent to calling **phpab** on the command-line like so:

```
phpab --output src/autoload.php --template build/src_autoload.php.in src
phpab --output tests/autoload.php --template build/tests_autoload.php.in tests
```

Invoking Ant in the directory that holds our *build.xml* file will now run the **build** target and its dependencies:

```
ant
Buildfile: /home/sb/bankaccount/build.xml

phpab:
  [exec] Autoload file 'src/autoload.php' generated.
  [exec]
  [exec] Autoload file 'tests/autoload.php' generated.
  [exec]

build:

BUILD SUCCESSFUL
Total time: 0 seconds
```

A useful option for Ant is `-projecthelp`. It prints project help information based on the `description` attribute of the `target` elements in the *build.xml*:

```
ant -projecthelp
Buildfile: /home/sb/bankaccount/build.xml
```

```
Main targets:
```

```
  phpab  Generate autoloader scripts
Default target: build
```

Targets that do not have a `description` attribute are considered private and are excluded from the `-projecthelp` output.

Setting Up Jenkins

Installing the PHP Quality Assurance Toolchain

All components and tools of the PHP quality assurance toolchain should be installed using the PEAR Installer, the backbone of the PHP Extension and Application Repository that provides a distribution system for PHP packages.



Depending on your OS distribution and/or your PHP environment, you may need to install PEAR or update your existing PEAR installation before you can proceed with the instructions in this chapter.

`sudo pear upgrade PEAR` usually suffices to upgrade an existing PEAR installation. <http://pear.php.net/manual/en/installation.getting.php> explains how to perform a fresh installation of PEAR.

The following two commands are all that is required to install the PHP quality assurance toolchain using the PEAR Installer:

```
pear config-set auto_discover 1
pear install pear.phpqatools.org/phpqatools PHPDocumentor
```

After the installation you can find the source files for the installed packages inside your local PEAR directory; the path is usually `/usr/lib/php`.

Here is an overview of what the tools we just installed are used for:

- [PHPUnit](#) is the de-facto standard for the unit testing of PHP code.
- [PHP_CodeSniffer](#) is the most commonly used tool for static analysis of PHP code. It is typically used to detect violations of code formatting standards but also supports software metrics as well as the detection of potential defects.
- [phpcpd \(PHP Copy/Paste Detector\)](#) searches for duplicated code in a PHP project.
- [PHP_Depend](#) is a tool for static code analysis of PHP code that is inspired by JDepend.

- [*phpmd \(PHP Mess Detector\)*](#) allows the definition of rules that operate on the raw data collected by PHP_Depend.
- [*phploc*](#) measures the scope of a PHP project by, among other metrics, means of different forms of the Lines of Code (LOC) software metric.
- [*PHP_CodeBrowser*](#) is a report generator that takes the XML output of the aforementioned tools as well as the sourcecode of the project as its input.
- Although it is currently being replaced by more modern tools such as [*phpdox*](#), we will use [*PHPDocumentor*](#) for automated API documentation generation for PHP code in this book.

In later chapters we will look at each of these tools individually and see how and why they are useful in a continuous integration setup.

Installing Jenkins

The Jenkins project provides native packages for Windows, Debian, Ubuntu, Red Hat, Fedora, CentOS, MacOS X, openSUSE, FreeBSD, OpenBSD, and Gentoo. Alternatively, you can manually install Jenkins into a directory of your choice.

The following steps detail how Jenkins can be installed on a UNIX system into the `/usr/local/jenkins` directory:

```
mkdir /usr/local/jenkins
cd /usr/local/jenkins
wget http://mirrors.jenkins-ci.org/war-stable/latest/jenkins.war
```



Using the web application archive (WAR) from the URL above we install an "older but stable" release of Jenkins.

The Jenkins developers produce a new release weekly to deliver bug fixes and new features rapidly to users and plugin developers who need them. But for more conservative users, it is preferable to stick to a release line that changes less and only for important bug fixes, even if such a release line lags behind in terms of features.

Please see the [Jenkins wiki](#) for more information about the different release lines.

We can now start up the Jenkins service:

```
export JENKINS_HOME=/usr/local/jenkins
java -jar jenkins.war
```

The web-based user interface of Jenkins is now available at `http://localhost:8080/`.

It is common to run Jenkins behind a reverse proxy. This can be achieved using [Apache HTTPD](#), for instance, by simply adding the following lines to your `httpd.conf` configuration file:

```
ProxyRequests Off
```

```
<Proxy *>  
    Order deny,allow  
    Allow from all  
</Proxy>
```

```
ProxyPass / http://127.0.0.1:8080/  
ProxyPassReverse / http://127.0.0.1:8080/  
ProxyMaxForwards 2
```

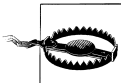
A configuration such as the above allows to restrict access to Jenkins by reusing existing access control lists you may already have in place for your webserver.

In addition to the web-based user interface, Jenkins also provides a Command-line interface which we can download like so:

```
wget http://localhost:8080/jnlpJars/jenkins-cli.jar
```

We can now install the plugins for Jenkins that are required to integrate PHP projects:

```
java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin checkstyle  
java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin cloverphp  
java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin dry  
java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin htmlpublisher  
java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin jdepend  
java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin plot  
java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin pmd  
java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin violations  
java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin xunit
```



After starting up the Jenkins service for the first time it may take a few minutes until it has loaded the plugin information from the project's website. Until this information is available the installation of plugins will not work.

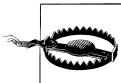
Here is an overview of what the plugins we just installed will be used for:

- The *Checkstyle* plugin is used to process the XML logfiles in Checkstyle format that PHP_CodeSniffer produces. Checkstyle is a tool that is commonly used in the Java world to help developers adhere to coding standards.
- The *Clover PHP* plugin is used to process the XML logfiles in Clover format that PHPUnit produces. Clover is a code coverage analysis tool that is commonly used in the Java world.
- The *DRY* plugin is used to process the XML logfiles in PMD-CPD format that phpcpd produces.
- The *HTML Publisher* plugin is used to publish the HTML files that are generated by PHP_CodeBrowser and PHPDocumentor.

- The *JDepend* plugin is used to process the XML logfiles in JDepend format that PHP_Depend produces. JDepend is a tool that is commonly used in the Java world to generate design quality metrics.
- The *Plot* plugin is used to plot the information gathered by phploc.
- The *PMD* plugin is used to process the XML logfiles in PMD format that the PHP Mess Detector (PHPMD) produces. PMD is a tool that is commonly used in the Java world to scan source code and detect potential problems such as possible dead code, duplicate code, or overcomplicated expressions.
- The *Violations* plugin is used to generate a summary report of the violations found by PHP_CodeSniffer, phpcpd, and PHPMD.
- The *xUnit* plugin is used to process the XML logfiles in JUnit format that PHPUnit produces.

The following two plugins are not required to integrate PHP projects with Jenkins. The *Git* plugin is required to interact with Git repositories and the *Green Balls* plugin customizes Jenkins to use a green ball instead of a blue ball to signify a successful build.

```
java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin git
java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin greenballs
```



If you want to use Jenkins with a version control system other than Git you will need to install the appropriate plugin instead of the aforementioned Git plugin.

Finally we schedule a restart of the Jenkins service for the changes to take effect:

```
java -jar jenkins-cli.jar -s http://localhost:8080 safe-restart
```

Figure 2-1 shows what Jenkins' dashboard looks like after the initial setup and configuration.

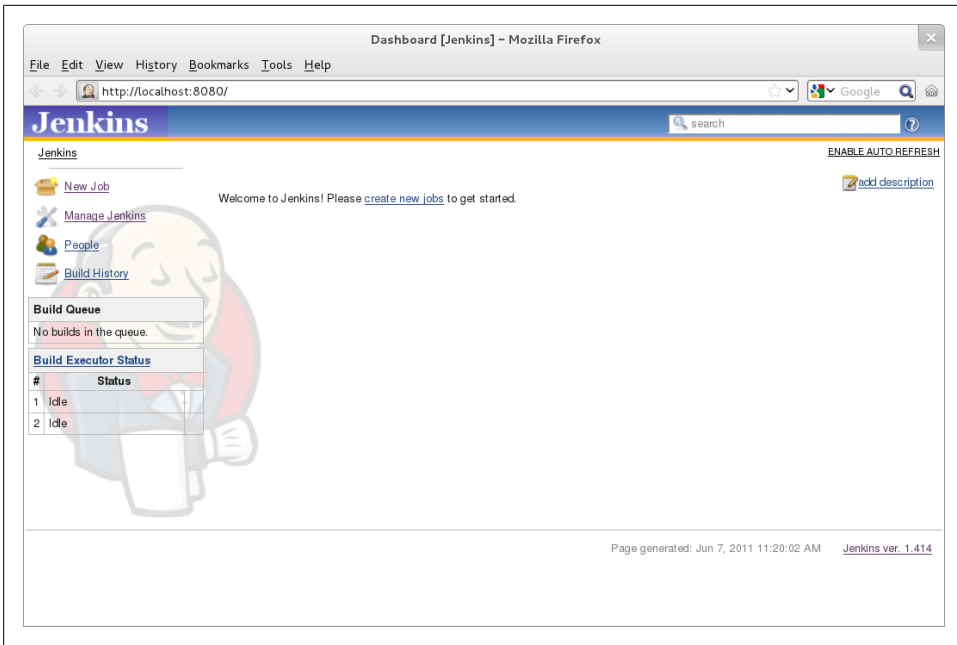


Figure 2-1. Dashboard view after initial setup and configuration

Continuous Integration

In this chapter we will expand our existing *build.xml* script for Ant to run unit tests using PHPUnit and then create a Jenkins job for our PHP project.

Running Unit Tests During the Build

Instead of putting this information into the *build.xml* script for Ant, we follow the concept of *Separation of Concerns* and use PHPUnit's configuration file feature to configure the tests we want to run, the logfiles (JUnit XML for test results and Clover XML for code coverage information) and reports (code coverage report in HTML format) we want generated, and other settings such as the bootstrap script (our autoloader).

[Example 3-1](#) shows the *phpunit.xml.dist* configuration file for our project.

Example 3-1. phpunit.xml.dist configuration file for PHPUnit

```
<?xml version="1.0" encoding="UTF-8"?>

<phpunit bootstrap="tests/autoload.php" backupGlobals="false"
    backupStaticAttributes="false" strict="true" verbose="true">
    <testsuite name="BankAccount">
        <directory suffix="Test.php">tests/unit</directory>
    </testsuite>
    <logging>
        <log type="coverage-clover" target="build/logs/clover.xml"/>
        <log type="coverage-html" target="build/coverage" title="BankAccount"/>
        <log type="junit" target="build/logs/junit.xml"/>
    </logging>
    <filter>
        <whitelist addUncoveredFilesFromWhitelist="true">
            <directory suffix=".php">src</directory>
            <exclude>
                <file>src/autoload.php</file>
            </exclude>
        </whitelist>
    </filter>
</phpunit>
```

Invoking **phpunit** without any parameters in the project directory (where the *phpunit.xml.dist* configuration file resides) will result in PHPUnit running the tests the way we set it up in the configuration.

We can now add a target to our *build.xml* script that invokes PHPUnit to run our unit tests. While we are at it, we add two other targets that handle cleaning up build artifacts and preparing directories for build artifacts, respectively:

- The **clean** target is responsible for cleaning up (deleting) any artifacts that may have been produced by a previous build. This target is a dependency of **prepare** but invoking it manually is also useful sometimes.
- The **prepare** target invokes the **clean** target and then creates the directories to which PHPUnit writes its logfiles and reports and (by means of the **phpab** target) generates the autoloader code.
- The **phpunit** target invokes PHPUnit.

[Example 3-2](#) shows how these three new targets are implemented. The implementation of the **phpab** task remains the same as shown in [Chapter 1](#) and is not repeated here.

Example 3-2. build.xml script that invokes PHPUnit

```
<?xml version="1.0" encoding="UTF-8"?>

<project name="BankAccount" default="build">
  <target name="build" depends="prepare,phpunit"/>

  <target name="clean" description="Cleanup build artifacts">
    <delete dir="${basedir}/build/coverage"/>
    <delete dir="${basedir}/build/logs"/>
  </target>

  <target name="prepare" depends="clean,phpab" description="Prepare for build">
    <mkdir dir="${basedir}/build/coverage"/>
    <mkdir dir="${basedir}/build/logs"/>
  </target>

  <target name="phpab" description="Generate autoloader scripts">
    <!-- ... -->
  </target>

  <target name="phpunit" description="Run unit tests with PHPUnit">
    <exec executable="phpunit" failonerror="true"/>
  </target>
</project>
```

[Example 3-3](#) shows the output of running the *build.xml* script with Ant.

Example 3-3. Running the build.xml script

```
ant
Buildfile: /home/sb/bankaccount/build.xml
```

```

clean:
[delete] Deleting directory /home/sb/bankaccount/build/coverage
[delete] Deleting directory /home/sb/bankaccount/build/logs

phpab:
[exec] Autoload file '/home/sb/bankaccount/src/autoload.php' generated.
[exec]
[exec] Autoload file '/home/sb/bankaccount/tests/autoload.php' generated.
[exec]

prepare:
[mkdir] Created dir: /home/sb/bankaccount/build/coverage
[mkdir] Created dir: /home/sb/bankaccount/build/logs

phpunit:
[exec] PHPUnit 3.5.15 by Sebastian Bergmann.
[exec]
[exec] .....
[exec]
[exec] Time: 0 seconds, Memory: 9.50Mb
[exec]
[exec] OK (39 tests, 69 assertions)
[exec]
[exec] Writing code coverage data to XML file, this may take a moment.
[exec]
[exec] Generating code coverage report, this may take a moment.
[exec]

build:

BUILD SUCCESSFUL
Total time: 2 seconds

```

We should make sure that the PHP code does not contain any syntax errors before we try to run the unit tests. The code below implements a target that uses Apache Ant's `<apply>` task to invoke PHP's lint checker (`php -l`) for each source code file in the `src` and `tests` directories. The `<apply>` task works like the `<exec>` task but can operate on multiple files when provided with a `<fileset>`.

```

<target name="lint">
  <apply executable="php" failonerror="true">
    <arg value="-l" />

    <fileset dir="${basedir}/src">
      <include name="**/*.php" />
    </fileset>

    <fileset dir="${basedir}/tests">
      <include name="**/*.php" />
    </fileset>
  </apply>
</target>

```

Now we have everything in place to create a job for our PHP project in Jenkins.

Creating a Jenkins Job

We start by accessing the the web-based user interface of Jenkins that is available at <http://localhost:8080/> and clicking on the *New Job* link in the upper left corner. This will start Jenkins' wizard to create a new job. On its first page, the wizard asks us for a name for the project and the type of project we want to build (see [Figure 3-1](#)). We choose *free-style software project* as this allows us to configure a job that can combine any version control system with any build system.

New Job [Jenkins] - Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://localhost:8080/view/All/newJob

Jenkins search

Jenkins » All

New Job

Job name

☒ **Build a free-style software project**
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

☐ **Build a maven2/3 project**
Build a maven2 project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

☐ **Build multi-configuration project**
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

☐ **Monitor an external job**
This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system. See [the documentation for more details](#).

Build Queue
No builds in the queue.

Build Executor Status

#	Status
1	Idle
2	Idle

Page generated: Jul 1, 2011 11:54:31 AM Jenkins ver. 1.414

Figure 3-1. Creating a new Jenkins job

The second (and final) page of the wizard we configure the details of the job including what to build as well as how and when to build it.

In the *Source Code Management* section (see [Figure 3-2](#)) of the form we configure the version control system we want to use. We are using Git for our example and fill in the repository's URL (*git://github.com/thePHPcc/bankaccount.git*) as well as the name of the branch we want to integrate (*master*).

Source Code Management

☐ CVS

☒ Git

Repositories

URL of repository

Advanced...

Delete Repository

Add

Branches to build

Branch Specifier (blank for default):

Delete Branch

Add

Advanced...

Repository browser

☐ None

☐ Subversion

Figure 3-2. Configuring the version control system

In the *Build Triggers* section (see [Figure 3-3](#)) of the form we configure when Jenkins should build our job. There are three possible triggers for building a job:

Polling the Version Control System

A job can be configured to be built when changes to the source code are detected. The version control system will be periodically polled for the required information.

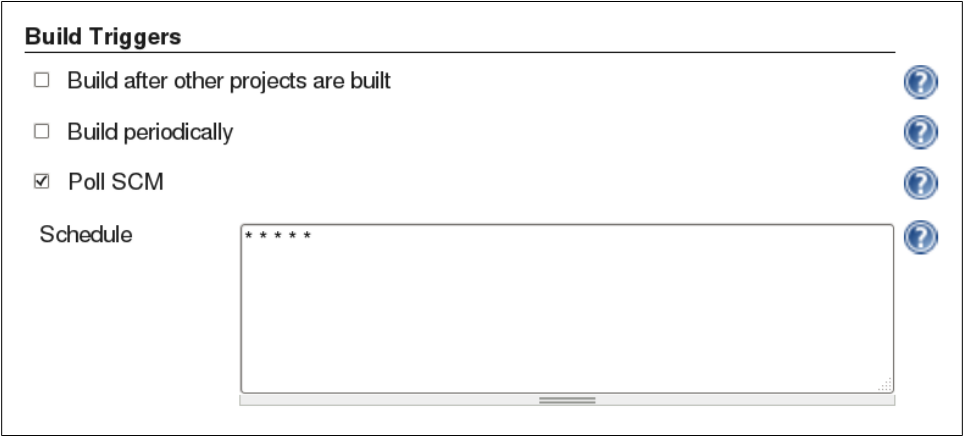
Building at specific times

A job can be configured to be built at specific times or intervals, for instance every night or every two hours.

Triggered by build of other job

A job can be configured to depend on another job. A new build is triggered when a build of that other job finishes.

We configure our job to poll the Git repository every minute. As Jenkins uses the syntax (with minor differences) of cron, the time-based scheduler in Unix-like operating systems, to specify build intervals, we fill in `* * * * *` in the *Schedule* field.



Build Triggers

☐ Build after other projects are built

☐ Build periodically

☒ Poll SCM

Schedule

Figure 3-3. Configuring the build triggers

Now that we have configured *what* to build and *when* build it, the next step is to configure *how* to build it. Since we are using Apache Ant for building our project this is really easy, as can be seen in [Figure 3-4](#).

The only things left to do are configuring the two reports, test results and code coverage, that we want Jenkins to publish. For this we simply need to configure the locations of the respective build artifacts (`build/logs/junit.xml`, `build/logs/cover.xml`, and `build/coverage`) as can be seen in [Figure 3-5](#) and [Figure 3-6](#).

Build

Invoke Ant

Ant Version

Default

Targets

Advanced...

Delete

Add build step

Figure 3-4. Configuring the build

☒ Publish testing tools result report

PHPUnit-3.4 (default) Pattern

build/logs/junit.xml

Fail the build if test results were not updated this run

Delete temporary JUnit files

Stop and set the build to 'failed' status if there are errors when processing a result file

Delete

Figure 3-5. Configuring the test report

☒ Publish Clover PHP Coverage Report

Clover XML Location

build/logs/clover.xml

Specify the name of the Clover xml file generated relative to [the workspace root](#).

☒ Publish HTML Report

Clover HTML report directory

build/coverage

Specify the path to the directory that contains the Clover HTML report file, relative to [the workspace root](#)

☐ Disable report archiving

Figure 3-6. Configuring the code coverage report

Figure 3-7 through Figure 3-11 show how Jenkins reports the information collected during the build.

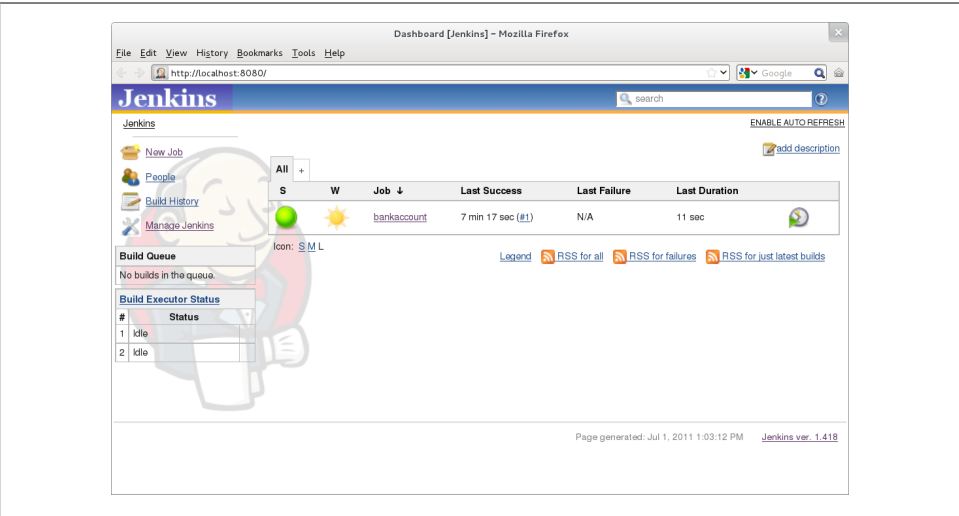


Figure 3-7. Dashboard view after the initial build of our project

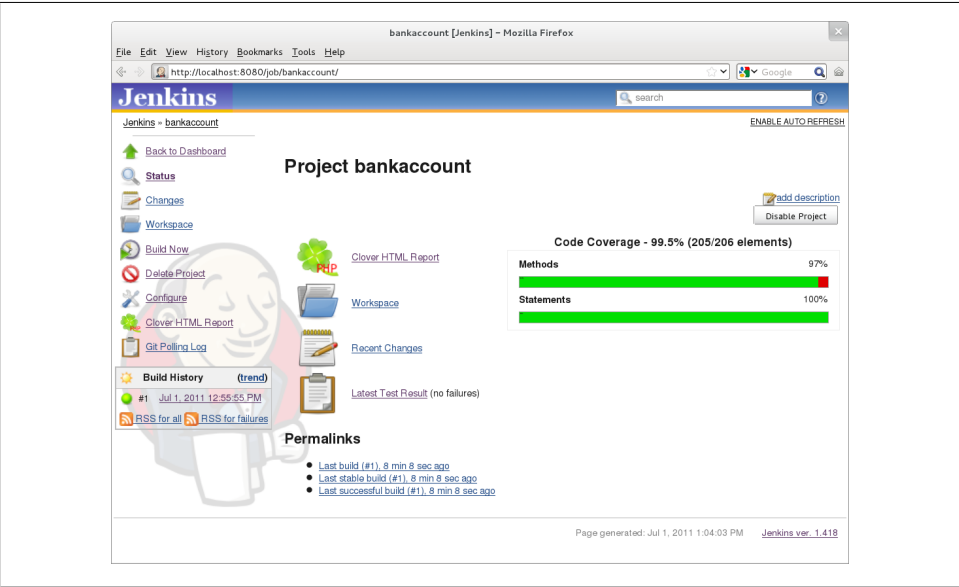


Figure 3-8. Project overview after the initial build of our project

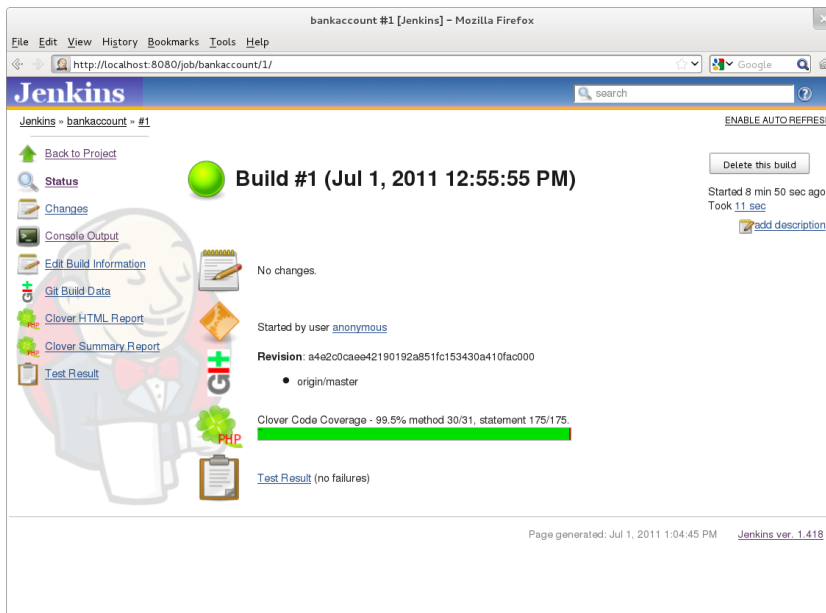


Figure 3-9. Summary information after the initial build of our project

bankaccount #1 (root) [Jenkins] - Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://localhost:8080/job/bankaccount/1/testReport/(root)/

Jenkins

Jenkins » bankaccount » #1 » Test Results » (root)

ENABLE AUTO REFRESH

Back to Project

Status

Changes

Console Output

Edit Build Information

History

Git Build Data

Clover HTML Report

Clover Summary Report

Test Result

Test Result : (root)

0 failures

39 tests

Took 0.27 sec

add description

All Tests

Class	Duration	Fail	(diff)	Skip	(diff)	Total	(diff)
BankAccountControllerTest	29 ms	0		0		3	+3
BankAccountListControllerTest	46 ms	0		0		1	+1
BankAccountListViewTest	5 ms	0		0		1	+1
BankAccountMapperTest	0.13 sec	0		0		9	+9
BankAccountTest	5 ms	0		0		5	+5
BankAccountViewTest	8 ms	0		0		1	+1
ControllerFactoryTest	13 ms	0		0		3	+3
FrontControllerTest	2 ms	0		0		1	+1
HashMapTest	3 ms	0		0		3	+3
MapperFactoryTest	3 ms	0		0		2	+2
RequestTest	3 ms	0		0		3	+3
ResponseTest	2 ms	0		0		2	+2

Figure 3-10. Test results for the initial build of our project

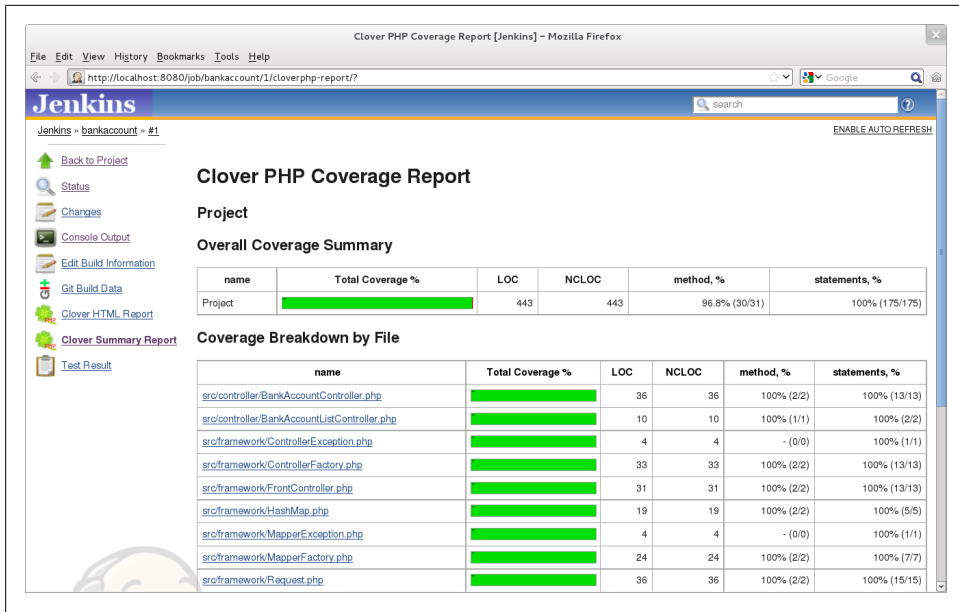


Figure 3-11. Code Coverage report for the initial build of our project

Continuous Inspection

The practice of Continuous Inspection expands on the ideas of Continuous Integration by performing an automated code review each time the code is changed. This makes it possible to detect undesirable developments such as increasing code complexity as early as possible and to counter them before it becomes too expensive. As Paul Duvall says:

An inspection at every change keeps defects in range.

In this chapter we will extend our existing *build.xml* script for Ant as well as our Jenkins job to invoke tools that, among other things, generate API documentation, calculate software metrics, and look for coding standard violations and duplicate code.

API Documentation

Well-written object-oriented code basically documents itself. Tools such as [PHPDocu](#)
[mentor](#) extract this information and render it in a useful format such as HTML. [Example 4-1](#) shows how to invoke PHPDocumentor from our Apache Ant build script.

Example 4-1. The `phpdoc` build target for PHPDocumentor

```
<target name="phpdoc"
  description="Generate API documentation using PHPDocumentor">
  <exec executable="phpdoc">
    <arg value="--directory" />
    <arg path="${basedir}/src" />
    <arg value="--target" />
    <arg path="${basedir}/build/api" />
  </exec>
</target>
```

Software Metrics

[phploc](#) can be used to track project size metrics, such as Lines of Code (LOC), over time. [Example 4-2](#) shows how to invoke it from our Apache Ant build script.

Example 4-2. The phploc build target for PHPLOC

```
<target name="phploc" description="Measure project size using PHPLOC">
  <exec executable="phploc">
    <arg value="--log-csv" />
    <arg value="${basedir}/build/logs/phploc.csv" />
    <arg path="${basedir}/src" />
  </exec>
</target>
```

In the above, we let PHPLOC write its data to a CSV file. The data from that file can be plotted using Jenkins' Plot plugin.

As software developers our focus is generally not on the *external aspects* of software quality such as functionality and usability. Instead we care about the *internal aspects* of software quality. This means that we are interested in readable code that is easy to understand, adapt, and extend. Implementing new features (or changing existing ones) will become more and more difficult and thus expensive over time if the internal quality of the software is neglected.

Internal software quality is measured through software metrics. A software metric is, in general, a function that maps a software unit onto a numeric value. The Cyclomatic Complexity and NPath Complexity software metrics measure, for instance, the complexity of a unit of code.

The cyclomatic complexity is the number of possible decision paths in a unit of code, usually a method or class. It is calculated by counting the control structures and boolean operators in a program unit and represents the structural complexity of a program unit. The idea is that a sequence of commands is easier to understand than a branch in the control flow.

A large cyclomatic complexity indicates that a program unit is susceptible for defects and hard to test. The more execution paths a program unit has, the more tests are required. The NPath complexity counts the number of acyclic execution paths and provides a lower bound for the amount of unit testing required for the unit of code.

[PHP_Depend](#) is the tool of choice to calculate a wide variety of software metrics for PHP code. [Example 4-3](#) shows how to invoke it from our Apache Ant build script.

Example 4-3. The `pdepend` build target for `PHP_Depend`

```
<target name="pdepend"
    description="Calculate software metrics using PHP_Depend">
  <exec executable="pdepend">
    <arg value="--jdepend-xml=${basedir}/build/logs/jdepend.xml" />
    <arg value="--jdepend-chart=${basedir}/build/pdepend/dependencies.svg" />
    <arg value="--overview-pyramid=${basedir}/build/pdepend/overview-pyramid.svg" />
    <arg path="${basedir}/src" />
  </exec>
</target>
```

Figure 4-1 shows how to configure a post-build action in our Jenkins job to publish the results of the analysis performed by `PHP_Depend`.

☒ Report JDepend

Pre-generated JDepend File

Provide a path to a JDepend file created during the build.
 Use a preceding "/" to specify an absolute path, leave off the "/" to specify a path within the workspace.
 Leave blank to have the plugin generate its own file.

Figure 4-1. Post-Build Action: Publish JDepend analysis results

Figure 4-2 and Figure 4-3 show examples of the visualizations generated by `PHP_Depend`. These can be displayed on the project overview page of Jenkins by putting the HTML code below into the project description:

```
<embed height="300"
  src="http://localhost:8080/job/job-name/ws/build/pdepend/overview-pyramid.svg"
  type="image/svg+xml"
  width="500"/>

<embed height="300"
  src="http://localhost:8080/job/job-name/ws/build/pdepend/dependencies.svg"
  type="image/svg+xml"
  width="500"/>
```

In the above, you need to replace `job-name` with the name of the Jenkins job. In our example this would be `bankaccount`.

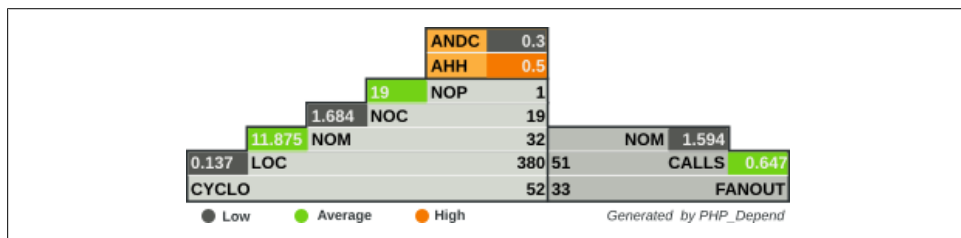


Figure 4-2. Software Metrics Overview Pyramid generated by `PHP_Depend`

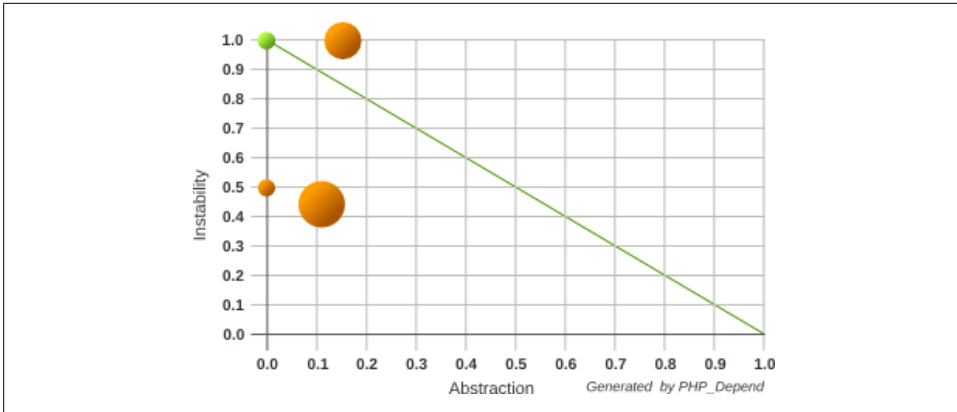


Figure 4-3. Dependencies chart generated by PHP_Depend

Duplicate Code

A class that does too much and has no clear responsibility, is *"a splendid breeding place for duplicated code, chaos and death"* (Martin Fowler). Duplicated code makes software maintenance more difficult, since all duplicates of one piece of code must be kept consistent, and a defect that has been found in duplicated code cannot be fixed in just one spot.

The [PHP Copy/Paste Detector](#) can be used to automatically detect duplicated code in a PHP project. [Example 4-4](#) shows how to invoke it from our Apache Ant build script.

Example 4-4. The phpcpd build target for the PHP Copy/Paste Detector

```
<target name="phpcpd" description="Find duplicate code using PHPCPD">
  <exec executable="phpcpd">
    <arg value="--log-pmd" />
    <arg value="${basedir}/build/logs/pmd-cpd.xml" />
    <arg path="${basedir}/src" />
  </exec>
</target>
```

[Figure 4-4](#) shows how to configure a post-build action in our Jenkins job to publish the results of the analysis performed by the PHP Copy/Paste Detector.

☒ Publish duplicate code analysis results ?

Duplicate code results

[Fileset includes](#) setting that specifies the generated raw XML report files, such as `**/cpd.xml` or `**/simian.xml`. Basedir of the fileset is [the workspace root](#). If no value is set, then the default `**/cpd.xml` is used. Be sure not to include any non-report files into this pattern.

High priority threshold

Minimum number of duplicated lines for high priority warnings.

Normal priority threshold

Minimum number of duplicated lines for normal priority warnings.

[Advanced...](#)

Figure 4-4. Post-Build Action: Publish duplicate code analysis results



[Started by an SCM change](#)



Revision: a4e2c0caee42190192a851fc153430a410fac000

- origin/master



Checkstyle: 0 warnings from one Checkstyle file.

- No warnings since build 1.
- New zero warnings highscore: no warnings since yesterday!



PMD: [1 warning](#) from one PMD file.

- [1 new warning](#)



Duplicate Code: 0 warnings from one analysis file.

- No warnings since build 1.
- New zero warnings highscore: no warnings since yesterday!



Clover Code Coverage - 99.5% method 30/31, statement 175/175.



[Test Result](#) (no failures)

Figure 4-5. Summary information for the build

Coding Standard Violations

In a software development project it is important that the team adheres to a single coding standard. A common coding standard lets you focus on solving problems that matter instead of wasting time understanding code that was written using a different style.

PHP_CodeSniffer is the tool of choice to detect violations of code formatting standards. It ships with hundreds of *sniffs* that each check for one particular code property. You can define your own rule set by picking and choosing the sniffs you need or use one of the built-in rule sets such as PEAR or Zend.

Example 4-5 shows an excerpt of the *build/phpcs.xml* configuration file for *PHP_CodeSniffer* that is used in the example project. In this excerpt, we define a coding standard that demands the opening curly brace of a class, function, or method to be on the next line, that disallows the usage of tabs to indent scopes, and that ensures proper indenting of scopes.

Example 4-5. *build/phpcs.xml* configuration file for *PHP_CodeSniffer*

```
<?xml version="1.0"?>
<ruleset name="MyRuleset">
  <description>My rule set for PHP_CodeSniffer</description>

  <rule ref="Generic.Functions.OpeningFunctionBraceBsdAllman"/>
  <rule ref="Generic.WhiteSpace.DisallowTabIndent"/>
  <rule ref="Generic.WhiteSpace.ScopeIndent"/>
</ruleset>
```

Example 4-6 shows how to invoke *PHP_CodeSniffer* from our Apache Ant build script.

Example 4-6. The *phpcs* build target for *PHP_CodeSniffer*

```
<target name="phpcs"
  description="Find coding standard violations using PHP_CodeSniffer">
  <exec executable="phpcs" output="/dev/null">
    <arg value="--report=checkstyle" />
    <arg value="--report-file=${basedir}/build/logs/checkstyle.xml" />
    <arg value="--standard=${basedir}/build/phpcs.xml" />
    <arg path="${basedir}/src" />
  </exec>
</target>
```

Figure 4-6 shows how to configure a post-build action in our Jenkins job to publish the results of the analysis performed by *PHP_CodeSniffer*.

Post-build Actions

☒ Publish Checkstyle analysis results

Checkstyle results:

[Fileset includes](#) setting that specifies the generated raw CheckStyle XML report files, such as `**/checkstyle-result.xml`. Basedir of the fileset is [the workspace root](#); if no value is set, then the default `**/checkstyle-result.xml` is used. Be sure not to include any non-report files into this pattern.

Advanced...

Figure 4-6. Post-Build Action: Publish Checkstyle analysis results

The *PHP Mess Detector* allows the definition of rules that operate on the raw data collected by PHP_Depend. Its focus is therefore not on the detection of code formatting violations but rather on issues such as possible bugs, hard-to-maintain code, unused parameters and variables as well as unused methods. It ships with about 30 rules that each check for one particular code property. You can define your own rule set by picking and choosing the rules you need or select one or more of the built-in rule sets.

Example 4-7 shows an excerpt of the *build/phpmd.xml* configuration file for PHPMD that is used in the example project. In this excerpt, we define a coding standard that enforces thresholds for the Cyclomatic Complexity and NPath Complexity software metrics, prohibits the usage of the `eval`, `exit`, and `goto` constructs of the PHP programming language, and that reports unused arguments and variables as well as unused private attributes and private methods.

Example 4-7. *build/phpmd.xml* configuration file for PHPMD

```
<?xml version="1.0"?>

<ruleset name="MyRuleset"
  xmlns="http://pmd.sf.net/ruleset/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://pmd.sf.net/ruleset/1.0.0 http://pmd.sf.net/
ruleset_xml_schema.xsd"
  xsi:noNamespaceSchemaLocation="http://pmd.sf.net/ruleset_xml_schema.xsd">
  <description>My rule set for PHPMD</description>

  <rule ref="rulesets/codesize.xml/CyclomaticComplexity" />
  <rule ref="rulesets/codesize.xml/NPathComplexity" />

  <rule ref="rulesets/design.xml/EvalExpression" />
  <rule ref="rulesets/design.xml/ExitExpression" />
  <rule ref="rulesets/design.xml/GotoStatement" />

  <rule ref="rulesets/unusedcode.xml/UnusedFormalParameter" />
  <rule ref="rulesets/unusedcode.xml/UnusedLocalVariable" />
  <rule ref="rulesets/unusedcode.xml/UnusedPrivateField" />
  <rule ref="rulesets/unusedcode.xml/UnusedPrivateMethod" />
</ruleset>
```

Example 4-8 shows how to invoke PHPMD from our Apache Ant build script.

Example 4-8. The `phpmd` build target for PHPMD

```
<target name="phpmd"
    description="Perform project mess detection using PHPMD">
  <exec executable="phpmd">
    <arg path="${basedir}/src" />
    <arg value="xml" />
    <arg value="${basedir}/build/phpmd.xml" />
    <arg value="--reportfile" />
    <arg value="${basedir}/build/logs/pmd.xml" />
  </exec>
</target>
```

Figure 4-7 shows how to configure a post-build action in our Jenkins job to publish the results of the analysis performed by PHPMD. Figure 4-8 shows an example of what this report looks like.

☒ Publish PMD analysis results

PMD results

build/logs/pmd.xml

Fileset includes setting that specifies the generated raw PMD XML report files, such as `**/pmd.xml`. Basedir of the fileset is the workspace root. If no value is set, then the default `**/pmd.xml` is used. Be sure not to include any non-report files into this pattern.

Advanced...

Figure 4-7. Post-Build Action: Publish PMD analysis results

PMD Result

Warnings Trend

All Warnings	New Warnings	Fixed Warnings
1	1	0

Summary

Total	High Priority	Normal Priority	Low Priority
1	0	1	0

Details

DetailsNew

BankAccountListController.php:4, UnusedFormalParameter, Priority: Normal

Avoid unused parameters such as `$request`.

Avoid passing parameters to methods or constructors and then not using those parameters.

public class Foo {
 private void bar(String howdy) {
 // howdy is not used
 }
}

Figure 4-8. PMD Result

Figure 4-9 shows how to configure a post-build action in our Jenkins job to publish a combined report of the violations found by PHP_CodeSniffer, PHP Copy/Paste Detector, and PHPMD.

☒ Report Violations

				XML filename pattern
checkstyle	10	999	999	build/logs/checkstyle.xml
codenarc	10	999	999	
cpd	10	999	999	build/logs/pmd-cpd.xml
findbugs	10	999	999	
fxcop	10	999	999	
gendarme	10	999	999	
jcreport	10	999	999	
jslint	10	999	999	
pmd	10	999	999	build/logs/pmd.xml

Figure 4-9. Post-Build Action: Report violations

Result Aggregation

[PHP_CodeBrowser](#) is a report generator that takes the XML logfiles generated by PHP_CodeSniffer, PHP Copy/Paste Detector, PHPMD, and PHPUnit as well as the sourcecode of the project as its input. The aggregated result of this is a browsable snapshot of the source code annotated with the violations found by these tools.

[Example 4-9](#) shows how to invoke PHP_CodeBrowser from our Apache Ant build script.

Example 4-9. The `phpcb` build target for `PHP_CodeBrowser`

```
<target name="phpcb"
  description="Aggregate tool output with PHP_CodeBrowser">
  <exec executable="phpcb">
    <arg value="--log" />
    <arg path="${basedir}/build/logs" />
    <arg value="--source" />
    <arg path="${basedir}/src" />
    <arg value="--output" />
    <arg path="${basedir}/build/code-browser" />
  </exec>
</target>
```

Figure 4-10 shows how to configure a post-build action in our Jenkins job to publish the HTML reports generated by PHP_CodeBrowser and PHPDocumentor.

HTML directory to archive	Index page[s]	Report title	Keep past HTML reports	
build/api	index.html	API Documentation	☒	Delete
build/code-browser	index.html	Code Browser	☒	Delete

Add

Figure 4-10. Post-Build Action: Publish HTML reports

Complete Build Script

Example 4-10 shows the complete *build.xml* script for our project.

Example 4-10. Complete *build.xml* script

```
<?xml version="1.0" encoding="UTF-8"?>

<project name="BankAccount" default="build">
  <target name="build"
    depends="prepare,lint,phpunit,parallelTasks,phpcb"/>

  <target name="clean" description="Cleanup build artifacts">
    <delete dir="${basedir}/build/api"/>
    <delete dir="${basedir}/build/code-browser"/>
    <delete dir="${basedir}/build/coverage"/>
    <delete dir="${basedir}/build/logs"/>
    <delete dir="${basedir}/build/pdepend"/>
  </target>

  <target name="prepare" depends="clean,phpab"
    description="Prepare for build">
    <mkdir dir="${basedir}/build/api"/>
    <mkdir dir="${basedir}/build/code-browser"/>
    <mkdir dir="${basedir}/build/coverage"/>
    <mkdir dir="${basedir}/build/logs"/>
    <mkdir dir="${basedir}/build/pdepend"/>
  </target>

  <target name="phpab" description="Generate autoloader scripts">
    <exec executable="phpab">
      <arg value="--output" />
      <arg path="${basedir}/src/autoload.php" />
      <arg value="--template" />
      <arg path="${basedir}/build/src_autoload.php.in" />
      <arg path="${basedir}/src" />
    </exec>

    <exec executable="phpab">
```

```

    <arg value="--output" />
    <arg path="{basedir}/tests/autoload.php" />
    <arg value="--template" />
    <arg path="{basedir}/build/tests/autoload.php.in" />
    <arg path="{basedir}/tests" />
  </exec>
</target>

<target name="lint">
  <apply executable="php" failonerror="true">
    <arg value="-l" />

    <fileset dir="{basedir}/src">
      <include name="**/*.php" />
    </fileset>

    <fileset dir="{basedir}/tests">
      <include name="**/*.php" />
    </fileset>
  </apply>
</target>

<target name="phpunit" description="Run unit tests with PHPUnit">
  <exec executable="phpunit" failonerror="true"/>
</target>

<target name="parallelTasks"
  description="Run code analysis tasks in parallel">
  <parallel threadCount="2">
    <sequential>
      <antcall target="pdepend"/>
      <antcall target="phpmd"/>
    </sequential>
    <antcall target="phpcpd"/>
    <antcall target="phpcs"/>
    <antcall target="phpdoc"/>
    <antcall target="phploc"/>
  </parallel>
</target>

<target name="pdepend"
  description="Calculate software metrics using PHP_Depend">
  <exec executable="pdepend">
    <arg value="--jdepend-xml={basedir}/build/logs/jdepend.xml" />
    <arg value="--jdepend-chart={basedir}/build/pdepend/dependencies.svg" />
    <arg value="--overview-pyramid={basedir}/build/pdepend/overview-pyramid.svg" />
    <arg path="{basedir}/src" />
  </exec>
</target>

<target name="phpmd"
  description="Perform project mess detection using PHPMD">
  <exec executable="phpmd">
    <arg path="{basedir}/src" />
    <arg value="xml" />
  </exec>
</target>

```

```

    <arg value="${basedir}/build/phpmd.xml" />
    <arg value="--reportfile" />
    <arg value="${basedir}/build/logs/pmd.xml" />
  </exec>
</target>

<target name="phpcpd" description="Find duplicate code using PHPCPD">
  <exec executable="phpcpd">
    <arg value="--log-pmd" />
    <arg value="${basedir}/build/logs/pmd-cpd.xml" />
    <arg path="${basedir}/src" />
  </exec>
</target>

<target name="phploc" description="Measure project size using PHPLOC">
  <exec executable="phploc">
    <arg value="--log-csv" />
    <arg value="${basedir}/build/logs/phploc.csv" />
    <arg path="${basedir}/src" />
  </exec>
</target>

<target name="phpcs"
  description="Find coding standard violations using PHP_CodeSniffer">
  <exec executable="phpcs" output="/dev/null">
    <arg value="--report=checkstyle" />
    <arg value="--report-file=${basedir}/build/logs/checkstyle.xml" />
    <arg value="--standard=${basedir}/build/phpcs.xml" />
    <arg path="${basedir}/src" />
  </exec>
</target>

<target name="phpdoc"
  description="Generate API documentation using PHPDocumentor">
  <exec executable="phpdoc">
    <arg value="--directory" />
    <arg path="${basedir}/src" />
    <arg value="--target" />
    <arg path="${basedir}/build/api" />
  </exec>
</target>

<target name="phpcb"
  description="Aggregate tool output with PHP_CodeBrowser">
  <exec executable="phpcb">
    <arg value="--log" />
    <arg path="${basedir}/build/logs" />
    <arg value="--source" />
    <arg path="${basedir}/src" />
    <arg value="--output" />
    <arg path="${basedir}/build/code-browser" />
  </exec>
</target>
</project>

```

Automating the Automation

Over the course of the last two years I have successfully set up many Jenkins-based continuous integration environments for PHP projects. As I was going through the same manual steps (ironically, to set up an automated process) over and over again, I asked myself: would it not be nice if there were a standard for the build automation and continuous integration of PHP projects as well as tooling to support it?

Answering this question lead to the creation of two new open source projects that are the topic of this chapter: the *PHP Project Wizard* and the *Template for Jenkins Jobs for PHP Projects*.

PHP Project Wizard

The PHP Project Wizard (PPW) is a command-line tool that can be used to generate the scripts and configuration files necessary for the build automation of a PHP project.

The following two commands are all that is required to install the PHP Project Wizard using the PEAR Installer:

```
pear config-set auto_discover 1
pear install pear.phpunit.de/ppw
```

As you can see in [Example 5-1](#), the scripts and configuration files generated by the PHP Project Wizard can be configured using various command-line options.

Example 5-1. PHP Project Wizard's command-line options

```
ppw --help
```

PHP Project Wizard (PPW) 1.1.0 by Sebastian Bergmann.

Usage: ppw [switches] <directory>

```
--name <name>           Name of the project

--source <directory>    Directory with the project's sources (default: src)
--tests <directory>    Directory with the project's tests (default: tests)
For multiple directories use a comma separated list
```

```

--bootstrap <script>    PHPUnit bootstrap script (default: tests/autoload.php)
--phpcs <ruleset>       Ruleset for PHP CodeSniffer (default: build/phpcs.xml)
--phpmd <ruleset>       Ruleset(s) for PHPMD (default: build/phpmd.xml)

--apidoc-tool <tool>    Tool to use for API documentation (default: phpdoc)
Possible values are "phpdoc", "phpdox"

--disable-apidoc        Do not include API documentation in the build script
--disable-phpab         Do not include PHPAB in the build script

--force                Overwrite existing files

--help                 Prints this usage information
--version              Prints the version and exits

```

ppw --name bankaccount

PHP Project Wizard (PPW) 1.1.0 by Sebastian Bergmann.

```

Wrote build script for Apache Ant to /home/sb/bankaccount/build.xml
Wrote configuration for PHP_CodeSniffer to /home/sb/bankaccount/build/phpcs.xml
Wrote configuration for PHPMD to /home/sb/bankaccount/build/phpmd.xml
Wrote configuration for PHPUnit to /home/sb/bankaccount/phpunit.xml.dist
Copied templates for PHPAB to /home/sb/bankaccount/build

```

The only mandatory command-line option for `ppw` is `--name` which is used to set the name of the project. The tool is usually invoked in the project's root directory. By default, it expects the production code to be in a `src` directory and the test code to be in a `tests` directory.

[Example 5-2](#) shows the files generated by the PHP Project Wizard and [Example 5-3](#) lists the artifacts generated during the build.

These build artifacts should be excluded from version control and be added to `.gitignore`, for instance, to prevent developers from accidentally adding such files to the repository. The *Template for Jenkins Jobs for PHP Projects* which we discuss in the next section expects exactly these build artifacts in exactly these locations.

Example 5-2. Files generated by the PHP Project Wizard

```

.
├── build
│   ├── phpcs.xml
│   ├── phpmd.xml
│   ├── src_autoload.php.in
│   └── tests_autoload.php.in
├── build.xml
├── phpunit.xml.dist
├── src
│   ├── autoload.php
│   └── ...
├── tests
│   ├── autoload.php
│   └── ...

```

Example 5-3. PHP Project Wizard's command-line options

```
build
├── api
│   └── ...
├── code-browser
│   └── ...
├── coverage
│   └── ...
├── logs
│   ├── checkstyle.xml
│   ├── clover.xml
│   ├── jdepend.xml
│   ├── junit.xml
│   ├── pmd-cpd.xml
│   └── pmd.xml
├── pdepend
│   ├── dependencies.svg
│   └── overview-pyramid.svg
```

Template for Jenkins Jobs for PHP Projects

The *Template for Jenkins Jobs for PHP Projects* makes it easy to quickly set up a new job for a PHP project in Jenkins by removing the need to manually configure the post build actions. Here is how you use it:

1. Go into Jenkins' *jobs* directory and check out the *php-jenkins-template* project from its Git repository:

```
cd $JENKINS_HOME/jobs
git clone git://github.com/sebastianbergmann/php-jenkins-template.git php-template
chown -R jenkins:nogroup php-template/
```

2. Reload Jenkins' configuration, for instance using the Jenkins CLI:

```
java -jar jenkins-cli.jar -s http://localhost:8080 reload-configuration
```

3. Click on "New Job".
4. Enter a "Job name".
5. Select "Copy existing job" and enter "php-template" into the "Copy from" field.
6. Click "OK".
7. Replace "localhost:8080" with the hostname and port of your Jenkins installation and replace the two occurrences of "job-name" with the name of your job in the "Description" text box.
8. Disable the "Disable Build" option.
9. Fill in your "Source Code Management" information.
10. Configure a "Build Trigger", for instance "Poll SCM".
11. Click "Save".

Following these instructions is equivalent to following the manual configuration steps from [Chapter 3](#) and [Chapter 4](#).

Conclusion

Continuous Integration and Continuous Inspection as described in this book ideally bring together all components of the software system, source code and configuration alike, when a change set is committed to version control.

Continuous Integration automatically produces a known state of the software that can be verified using automated tests as well as static code analysis. When successful, the result of such a build, the *build artifact* represents a state of the software that is known to work correctly (at least all automated tests are satisfied) and can be made available for manual testing, for instance.

Continuous Inspection calculates software metrics that measure various aspects of the internal quality of the software for each build. Looking at this data over time facilitates a deeper understanding of the quality of the software throughout its lifecycle. This makes it possible to see trends and detect undesirable developments such as increasing code complexity as early as possible and to counter them before it becomes expensive. The reports generated by Continuous Inspection can be used by the development team when explaining technical debt and the need for a refactoring to their management: "In the last sprint we did not develop any new features but we cleaned up our code base to eliminate duplicate code, reduce complexity, and fix coding standards violations as you can see in these charts. Thanks to this refactoring we will be able to deliver new features faster and more reliable in the future."

When you have followed the instructions provided in this book then you have a state-of-the-art Continuous Integration and Continuous Inspection environment for your PHP projects in place. This section will give you some food for thought with regard to additional measures that you should explore and might want to implement.

Continuous Integration and Development Branches

Modern version control systems such as Git make the work with multiple branches efficient and simple. A common process when using Git is to have two main development branches named *master* and *development* that are used as follows:

- **master**
 - There is no active development in this branch
 - There are no direct commits into this branch
 - Changes to this branch are only merges from the **development** branch (or from hotfix branches, see below)
 - The state of the software must be stable in this branch
- **development**
 - This is the branch where active development takes place
 - It is considered best practice to develop new features in so called feature branches that are branched off of the **development**.
 - The state of the software may be unstable (for short periods of time) in this branch

When the time for a release has come you want to stabilize the state of the software in the **development** branch and then merge that branch into the **master** branch and create a release tag there, for instance.

When a problem in the **master** branch is found the developers should focus their efforts on fixing this problem. This should happen in a hotfix branch that is branched off of the **master** branch. Once the problem is fixed this hotfix branch can be merged back into the **master** branch and from there into the **development** branch. In case the bug fix increases the technical debt of the project a new branch should immediately be branched off of the **development** branch in which the respective refactoring to clean up the code will take place.

Both branches, **master** and **development** should be continuously integrated. This can be implemented by configuring two jobs in Jenkins, one for each branch. These two jobs only differ with regard to the branch they operate on.

While feature branches should be short-lived (to reduce the risk of conflicts when merging them into the **development** branch) it may make sense to also integrate them continuously. This can be set up in such a way, for instance, that a script (invoked by hook in the version control system or via a cron job) automatically creates and deactivates jobs in Jenkins when a feature branch is created or merged and deleted, respectively. Thanks to Jenkins' remote API this can be implemented quite easily.

Additional Testing

So far we have only discussed running unit tests as part of the build. These are the most valuable kind of automated tests as they cannot only report that something is broken but are also able to provide information where something is broken. This is possible because unit tests test a unit of code in isolation from its dependencies (using stubbing and mocking, for instance).

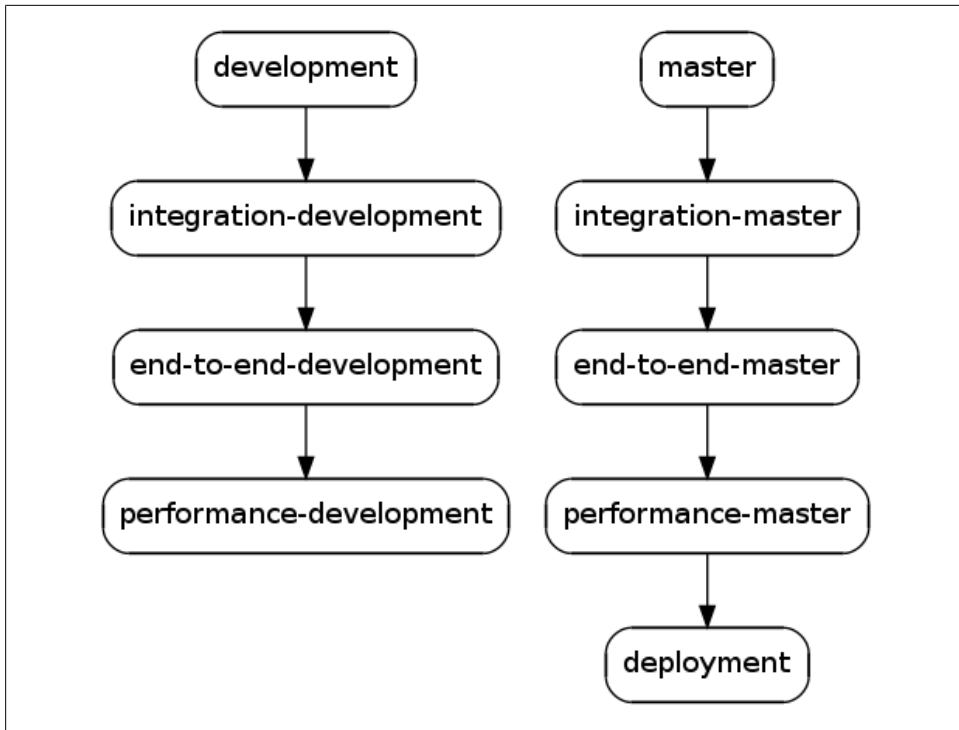


Figure 6-1. Build Pipeline with two development branches

In addition to unit tests you should also have automated integration tests ("larger" unit tests that do not isolate a unit of code from its dependencies), automated end-to-end tests (that use [Selenium](#), for instance, to instrument a real web browser to send real web requests to the web application deployed on a real webserver and test aspects of the software based on the real response) as well as automated performance tests (using [JMeter](#), for instance).

Instead of running all of these tests (unit tests, integration tests, end-to-end tests, performance tests) in one job for continuous integration you should set up a build pipeline of multiple jobs that depend on each other. As the tests grow in size, from small unit tests to large end-to-end tests, the resource usage required to run them increases. Against this background it makes no sense to run integration or end-to-end tests, for instance, when the unit tests already tell you that something is broken.

Figure 6-1 shows what such a build pipeline could look like.

Continuous Deployment

If you have enough confidence in your unit tests, integration tests, and end-to-end tests then you can consider automatically deploying your application after each successful build of the `master` branch.

Bibliography

- [Bergmann2011] Bergmann, Sebastian, and Stefan Priebisch. *Real-World Solutions for Developing High-Quality PHP Frameworks and Applications*. Wrox, 2011.
- [Duvall2007] Duvall, Paul. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.
- [Humble2010] Humble, Jez, and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley, 2010.
- [Smart2011] Smart, John Ferguson. *Jenkins: The Definitive Guide*. O'Reilly Media, 2011.

About the Author

Sebastian Bergmann is actively involved in the development of PHP and has created a wide range of tried-and-trusted development tools. As an internationally sought-after expert, he shares his knowledge and experience through widely read books and articles. His presentations at conferences around the world are intently followed by the PHP community and others.

The computer scientist (Diplom-Informatiker) is a co-founder of thePHP.cc and a pioneer in the field of quality assurance in PHP projects. His testing framework, PHPUnit, is a de facto standard.

