

PHPUnit Essentials

Table of Contents

Chapter 1: Installing PHPUnit	4
<hr/>	
Requirements	
Running PHP from the command line	
Composer – the dependency manager for PHP	
Installing Composer	
Installation	
Local installation	
System-wide installation	
Installing PEAR	
PHPUnit installation	
Other installation methods	
Installing the Linux package	
Manual installation	
Testing the installation	
Xdebug	
Installing Xdebug	
Summary	
Chapter 2: PHPUnit Support in IDEs	20
<hr/>	
IDEs and PHPUnit	
NetBeans	
Zend Studio	
Eclipse PDT	
Installing MakeGood	
Creating your FirstTest	
PhpStorm	
Summary	

Chapter 3: Tests and What They're All About

42

Understanding unit testing

What is a unit test?

Assertions

The importance of unit testing

Testing all possible scenarios

What makes a good test?

When to write tests

Anatomy of a PHPUnit test

Defining test methods

Testing functions

Testing methods

The MVC application architecture and tests

Testing controllers

Summary

Chapter 4: Database Testing

61

Which database to use

Tests for a database

DBUnit

Installing DBUnit

Database test cases

Datasets

Using DBUnit

Chapter 5: Continuous Integration	89
Using a Travis CI hosted service	
Setting up Travis CI	
Using Travis CI	
Using the Jenkins CI server	
Installation	
Usage	
Creating a job	
Results	
Using the Xinc PHP CI server	
Installation	
Usage	
Summary	

1

Installing PHPUnit

To be able to write and run PHPUnit tests, PHPUnit has to be installed on the machine. With the right approach, it is not difficult; however, in the past, this was a bit of an issue for many users because they were struggling with the right settings and all of the dependencies that needed to be installed.

Until recently, the recommended installation method for PHPUnit was by using PEAR; however, from PHPUnit 3.7 onwards, there is an option available to use **Composer**. Composer is a tool for dependency management in PHP, which makes PHP tools or libraries' installations much easier. There are a few more options to install PHPUnit, and this chapter describes them step by step.

Apart from what is required for PHPUnit installation, this chapter also describes the Xdebug installation process. This is optional, but this PHP extension really helps with processes such as remote debugging, code coverage analysis, and profiling PHP scripts. Later in the book, how to use Xdebug for debugging and generating code coverage will be explained, and it is more than an option for every PHP developer to have this extension.

Installing PHPUnit


This chapter describes how to install PHPUnit, and mentions what is required to be able to run PHPUnit tests; some of these tests are as follows:

- **PHP CLI:** This is a PHP command-line interface to run tests from the command line
- **PHPUnit:** This is a testing framework and tool for unit tests execution
- **Xdebug:** This is a PHP extension for remote debugging, code coverage, and much more

This chapter covers various PHPUnit installation methods, among which the easiest is the Composer or PEAR installation, but there are other installation methods available too. The following are the different PHPUnit installation methods:


- Composer installation
- PEAR installation
- Linux package installation
- Manual installation

As an extra step, it is recommended that you have Xdebug installed. Xdebug is a PHP extension that provides debugging and profiling capabilities and is also used by PHPUnit. Xdebug installation and configuration is described at the end of this chapter.

 On 21st of April, 2014, it was announced end of life for the PEAR installation method. PEAR installation should be available during the entire 2014. However, recommended installation methods are the Composer installation or PHAR archive.

Requirements

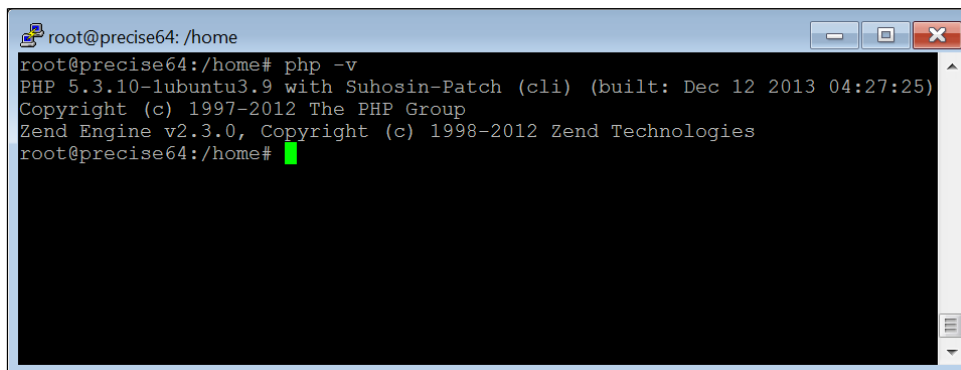
A developer who starts with PHPUnit is probably already working with PHP and has already installed the web server with PHP support such as Apache, NGINX, or IIS. It is strongly recommended that you use at least PHP 5.3.3 or higher; older PHP versions are not supported and maintained anymore.

 Information about PHPUnit and downloads and documentation can be found on the official PHPUnit site at <http://phpunit.de>.

PHPUnit is a unit testing framework written in PHP, but PHPUnit tests are not usually executed on a web server but through the PHP CLI – the PHP command-line interface. Through PHP CLI, the PHP script can be run as any other shell script.

Running PHP from the command line

As mentioned earlier, PHP CLI allows to run any PHP script from the command-line interface. Whichever tool is used to run PHPUnit tests, even if it has used the IDE, which seems to have everything preinstalled, it will always call PHP CLI in the background.



```

root@precise64: /home
root@precise64:/home# php -v
PHP 5.3.10-1ubuntu3.9 with Suhosin-Patch (cli) (built: Dec 12 2013 04:27:25)
Copyright (c) 1997-2012 The PHP Group
Zend Engine v2.3.0, Copyright (c) 1998-2012 Zend Technologies
root@precise64:/home#

```

You can install PHP CLI as a package (PHP5-CLI), or you can find it already installed with the web server package. Even software packages such as WAMP or XAMP have it there, but you might need to specify a path to web server binaries to environment variables to be able to run PHP from the command line anywhere on your machine.

To test if PHP CLI is installed on your machine, run the following command line:

```
> php -r "echo 'Hello, World!';"
```

This is a simple `Hello, World!` code executed and outputted by the command language interpreter, but it confirms PHP CLI is present and does what is expected. If you see an error message such as `command not found`, it implies PHP CLI is not installed or the system path is missing – a path to the PHP binary.

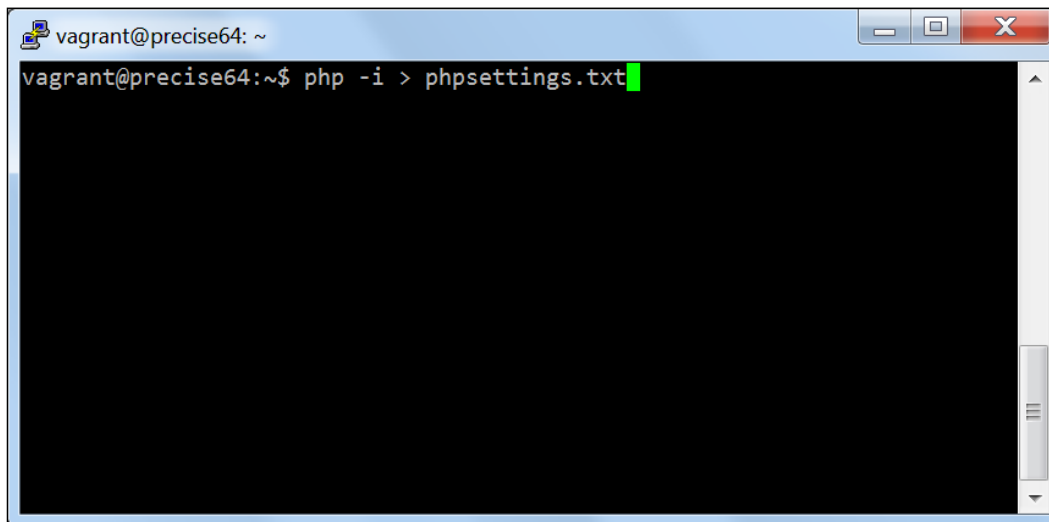


CLI may use a different `php.ini` configuration file than the one your web server uses. If so, you'll need to configure your PHP extensions and other bits and bobs in there separately. To check this, you can run the following command line:

```
> php -i
```

Installing PHPUnit

It will output a result of the `phpinfo()` function, but in a text version, where you will be able check configuration settings. If you have problem reading so many lines, try to output it in a text file, which you can open in a text editor, and search for the desired result. The output also can be captured in the text file, which can be easily read with any text editor.

A terminal window titled 'vagrant@precise64: ~' with standard window controls. The command prompt shows 'vagrant@precise64:~\$ php -i > phpsettings.txt' with a green cursor at the end of the line. The terminal background is black, and the text is white.

Composer – the dependency manager for PHP

Composer is a tool for dependency management in PHP. It allows you to declare the dependent libraries that the project needs, and it will install them. There is a configuration file called `composer.json`, which, as the name suggests, is a JSON format file. This is where you set the components/libraries and version your project will be using. When you run Composer, the Composer tool not only downloads all the required libraries, but also all the other required third-party libraries used. It also creates autoloader (the file to load all required classes), and all the hard work is done – everything that is required is ready to be used.

Composer is one of the best tools available for the PHP developers. It really helps and makes your life easier. It is something that was already available in other languages such as Maven for Java or Gems for Ruby and was missing in PHP. The best thing is, in this way, you can not only manage third-party libraries and packages, but also your own libraries.

Installing Composer

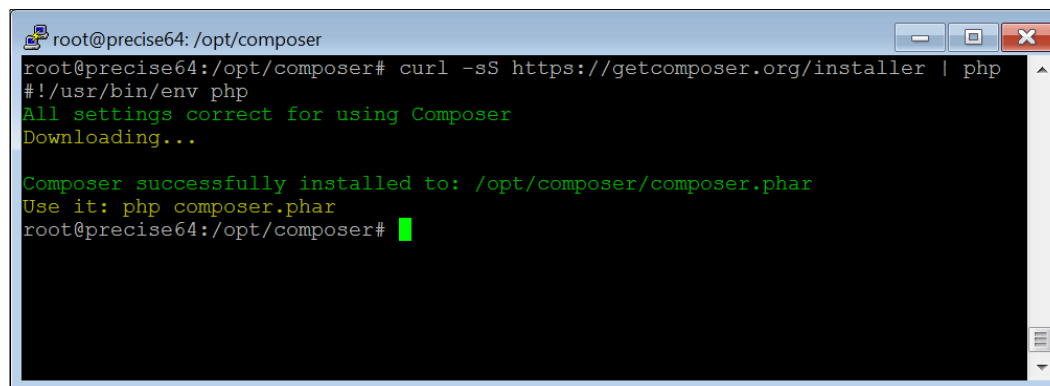
The Composer tool is just one file — `composer.phar`. The `.phar` file means a PHP archive (the PHAR file) similar to the JAR files of Java. All PHP files are packed in one archive file. You can download `composer.phar` from <http://getcomposer.org/download/> or by running the following command line:

```
curl -sS https://getcomposer.org/installer | php
```

If you haven't got curl installed, alternatively run the following command line:

```
>php -r "readfile('https://getcomposer.org/installer');" | php
```

As a result, you should see a message that `composer.phar` was downloaded, similar to the message displayed in the following screenshot:

A screenshot of a terminal window titled 'root@precise64: /opt/composer'. The terminal shows the execution of the command 'curl -sS https://getcomposer.org/installer | php'. The output is as follows: a prompt '#!/usr/bin/env php', the message 'All settings correct for using Composer' in green, 'Downloading...' in yellow, and 'Composer successfully installed to: /opt/composer/composer.phar' in green. It then says 'Use it: php composer.phar' in green. The prompt returns to 'root@precise64:/opt/composer#' with a green cursor. The terminal window has standard Linux window controls (minimize, maximize, close) in the top right corner.

```
root@precise64: /opt/composer
root@precise64:/opt/composer# curl -sS https://getcomposer.org/installer | php
#!/usr/bin/env php
All settings correct for using Composer
Downloading...

Composer successfully installed to: /opt/composer/composer.phar
Use it: php composer.phar
root@precise64:/opt/composer#
```

Installation

There are two options that are of the biggest advantage to Composer, compared to other installation methods. Composer gives you the flexibility to specify where to store PHPUnit and also where to store the command-line script so that you can have on your machine as many versions as you want, and by using Composer, you can still maintain and keep them all up to date.

Local installation

Here is one practical example. You can specify and download PHPUnit only for your specific project. This means that when your project was developed and tested, you were working with PHPUnit version 3.5; however, for the new project, you want to use version 3.7. For example, Zend Framework 1 uses PHPUnit version 3.5; however, applications are still developed, extended, and maintained on this system even when Zend Framework 2 is available. This is not a problem as each project will have its own version of PHPUnit with all the dependencies, and there is no need to worry about clashing versions or being stacked with dated versions.

For PHPUnit installation with Composer, you have to create a `composer.json` file. The simplest version could look the following lines of code:

```
{
    "require-dev": {
        "phpunit/phpunit": "3.7.*"
    }
}
```

System-wide installation

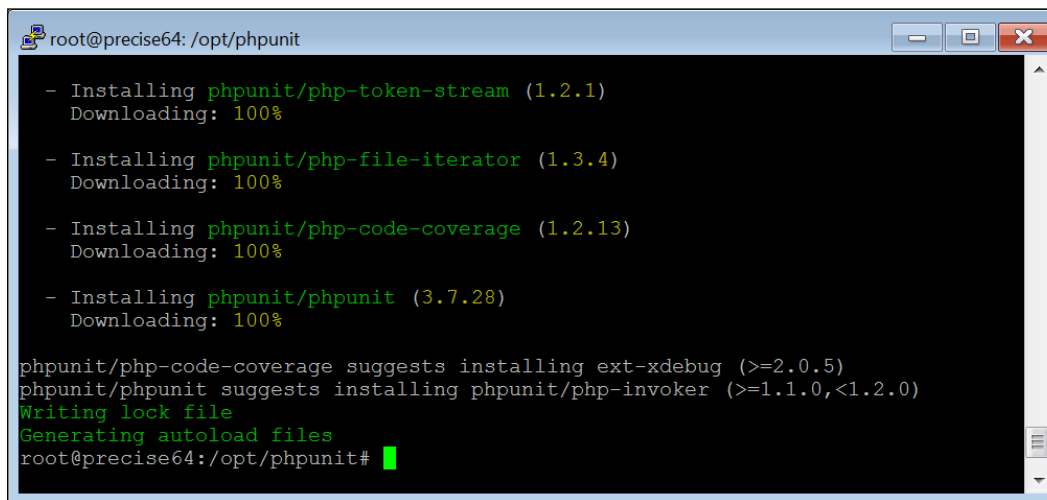
The second option is a system-wide installation. Well, basically it is the same thing, except it is created by a shell script / batch file to call PHPUnit from the command line. If a file is stored in a directory set in the system path, then it is available anywhere in your system. The following lines of code represent `composer.json`:

```
{
    "require": {
        "phpunit/phpunit": "3.7.*"
    },
    "config": {
        "bin-dir": "/usr/local/bin/"
    }
}
```

System-wide installation might be easier in the beginning, and allows you to run tests from the command line. It's easier to start with moving `composer.phar` to the project document root—the same place where the `composer.json` file is stored. Then, to install PHPUnit, just run the following command line:

```
> php composer.phar install
```

Composer explores the `composer.json` file and determines all the extra packages that need to be downloaded and installed. This should display a similar result as shown in the following screenshot:



```

root@precise64: /opt/phpunit

- Installing phpunit/php-token-stream (1.2.1)
  Downloading: 100%

- Installing phpunit/php-file-iterator (1.3.4)
  Downloading: 100%

- Installing phpunit/php-code-coverage (1.2.13)
  Downloading: 100%

- Installing phpunit/phpunit (3.7.28)
  Downloading: 100%

phpunit/php-code-coverage suggests installing ext-xdebug (>=2.0.5)
phpunit/phpunit suggests installing phpunit/php-invoker (>=1.1.0,<1.2.0)
Writing lock file
Generating autoload files
root@precise64:/opt/phpunit#

```

This means PHPUnit was successfully installed, including the required dependencies (YAML) and basic plugins. The line that suggests installing `ext-xdebug` means that you should install the PHP Xdebug extension, because it is a useful thing for development and also PHPUnit uses it to generate code coverage.

Installing PEAR

Another way to install PHPUnit is to use PEAR—the framework and distribution system for reusable PHP components. PEAR has been available since the early PHP 4 days, and more than a framework, it is just a set of useful packages and classes. Also, it is a distribution system for installing and keeping these packages up to date. The first step of installation is to install PEAR. Download the PHP archive file `go-pear.phar` from <http://pear.php.net/go-pear.phar>.

Similar to `composer.phar`, it is a PHP archive file that contains all that you need. Installation is simple, just run the following command line:

```
> php go-pear.phar
```

Follow the PEAR install instructions, hitting *Enter* to accept the default configuration as you go. After installation, you can test it by running PEAR from the command line by just running the following command line:

```
> pear
```

Installing PHPUnit

You should see something similar to the following message that will tell you where the PEAR script is stored:

The 'pear' command is now at your service at /root/pear/bin/pear

If it doesn't work, checking the installation message usually gives a clue as to where the problem is; this is shown in the following message:

```
** The 'pear' command is not currently in your PATH, so you need to
** use '/root/pear/bin/pear' until you have added
** '/root/pear/bin' to your PATH environment variable.
```

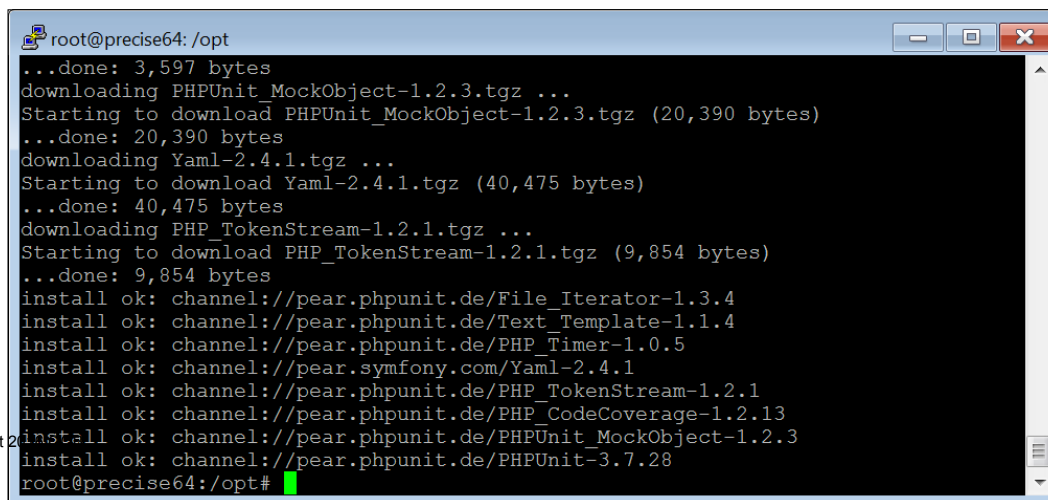
PEAR installation has one disadvantage compared to the Composer installation: it is a system-wide installation, which means you have just one version of PEAR—one version of PHPUnit installed in your system.

PHPUnit installation

When PEAR is successfully installed, PHPUnit installation is simple and straightforward, as shown in the following command line:

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

The first line means that PEAR checks and installs the required dependencies. The sources are called channels that will be automatically discovered and added. With older versions, you had to do it manually; however, it is not necessary anymore with `auto_discover`. The second line then installs the latest stable version of PHPUnit. The result should be similar to the following screenshot:



```
root@precise64: /opt
...done: 3,597 bytes
downloading PHPUnit_MockObject-1.2.3.tgz ...
Starting to download PHPUnit_MockObject-1.2.3.tgz (20,390 bytes)
...done: 20,390 bytes
downloading Yaml-2.4.1.tgz ...
Starting to download Yaml-2.4.1.tgz (40,475 bytes)
...done: 40,475 bytes
downloading PHP_TokenStream-1.2.1.tgz ...
Starting to download PHP_TokenStream-1.2.1.tgz (9,854 bytes)
...done: 9,854 bytes
install ok: channel://pear.phpunit.de/File_Iterator-1.3.4
install ok: channel://pear.phpunit.de/Text_Template-1.1.4
install ok: channel://pear.phpunit.de/PHP_Timer-1.0.5
install ok: channel://pear.symfony.com/Yaml-2.4.1
install ok: channel://pear.phpunit.de/PHP_TokenStream-1.2.1
install ok: channel://pear.phpunit.de/PHP_CodeCoverage-1.2.13
install ok: channel://pear.phpunit.de/PHPUnit_MockObject-1.2.3
install ok: channel://pear.phpunit.de/PHPUnit-3.7.28
root@precise64:/opt#
```

Other installation methods

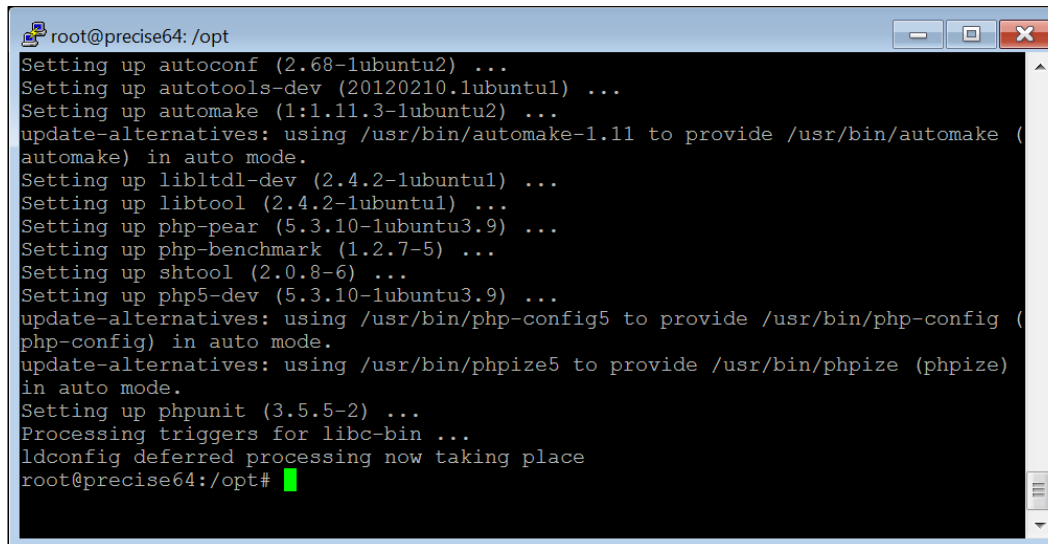
Along with the two methods that have been mentioned and recommended, there exist another two. It is up to you to choose the installation method that suits you the best (just be careful what you are installing and from where). If there is no specific reason, it's recommended choosing a stable version, and if possible the latest version. The clue could be which version is in the current PHPUnit manual, which can be found at <http://phpunit.de/manual/current/en/index.html>.

Installing the Linux package

For Linux users and users using Linux based virtual machines such as Vagrant and VirtualBox (Mac OS X, Windows), probably the fastest and least painful installation method is to use package manager. It depends on your Linux version, and the package management tool could be yum, apt or aptitude. Run the following command line:

```
>apt-get install phpunit
```

This will install all the required dependencies and libraries, and the result should be similar to the following screenshot:

A screenshot of a terminal window titled 'root@precise64: /opt'. The terminal displays the output of the command 'apt-get install phpunit'. The output shows the installation of various dependencies including autoconf, autotools-dev, automake, libltdl-dev, libtool, php-pear, php-benchmark, shtool, php5-dev, and finally phpunit (3.5.5-2). It also shows the processing of triggers for libc-bin and the deferred processing of ldconfig. The prompt 'root@precise64:/opt#' is visible at the bottom with a green cursor.

```
root@precise64: /opt
Setting up autoconf (2.68-1ubuntu2) ...
Setting up autotools-dev (20120210.1ubuntu1) ...
Setting up automake (1:1.11.3-1ubuntu2) ...
update-alternatives: using /usr/bin/automake-1.11 to provide /usr/bin/automake (
automake) in auto mode.
Setting up libltdl-dev (2.4.2-1ubuntu1) ...
Setting up libtool (2.4.2-1ubuntu1) ...
Setting up php-pear (5.3.10-1ubuntu3.9) ...
Setting up php-benchmark (1.2.7-5) ...
Setting up shtool (2.0.8-6) ...
Setting up php5-dev (5.3.10-1ubuntu3.9) ...
update-alternatives: using /usr/bin/php-config5 to provide /usr/bin/php-config (
php-config) in auto mode.
update-alternatives: using /usr/bin/phpize5 to provide /usr/bin/phpize (phpize)
in auto mode.
Setting up phpunit (3.5.5-2) ...
Processing triggers for libc-bin ...
ldconfig deferred processing now taking place
root@precise64:/opt#
```


Installing PHPUnit

If this is so easy, why is it not mentioned as the first installation method? Obviously, you can't use it if you are running PHP. It can't be used on Windows or Mac OS X, but there is something else. When this chapter was written, the latest available stable PHPUnit version for Composer and PEAR was 4.0. In comparison, for the Ubuntu Precise Pangolin package 3.5.5-2 version was available, Debian had version 3.6.10-1 for PHPUnit, and EPEL had PHPUnit-3.7.19-1.el6. This is something to watch out for. Some Linux distributions are very mature and very stable, but they stick to the package version that was available when the distribution was released. Then the distribution is supported for the next five years, but the package is never upgraded (just the security fixes are back-ported to the original version). This leads to situations where you have all the updates installed but still see hopelessly old versions of your tool/library.

Manual installation

The final option is manual installation. It is not too bad. There is one option that allows you to get PHPUnit with all that you need but without all the package management hassle. Just download the `phpunit.phar` archive from the PHPUnit site at <https://phar.phpunit.de/phpunit.phar>.

The package contains PHPUnit and all the required dependencies. To test, just run the following command line:

```
>php phpunit.phar -version
```

Or, you can create the `phpunit` executable by running the following commands:

```
>chmod +x phpunit.phar
>mv phpunit.phar /usr/local/bin/phpunit
```

Or, on Windows using `phpunit.bat` from the Git repository, which is also the last option, download PHPUnit source code directly by cloning the Git repository on GitHub by running the following command line:

```
>git clone https://github.com/sebastianbergmann/phpunit/
```

Testing the installation

When PHPUnit is installed, you should be able to run the PHPUnit tool from the following command line:

```
> phpunit
```

And then you should see a list of all the available options like the following list displayed:

PHPUnit 3.7.28 by Sebastian Bergmann.

```
Usage: phpunit [switches] UnitTest [UnitTest.php]
       phpunit [switches] <directory>
```

If it doesn't work, don't panic; try to think what might be wrong. Is the PHPUnit script on the system path? Have you installed it as user or root?

In the worst case scenario, you can try alternative installation methods or install it as a local installation just for one user. This means that all the files are going to be stored in the user home directory, and when you want to run PHPUnit, just go to `phpunit/bin` and there is the script. The difference with the system-wide installation is just that the files are stored in different places and PHPUnit is not added to the system path.

Anyway, now we would like to see a bit more. So, let's create our first tests and run it by using the freshly installed PHPUnit.

The best way to try PHPUnit is to create a simple unit test and execute it by using the freshly installed PHPUnit. Let's try basic mathematical operations to see if PHP counts correctly. Later, we will talk about tests syntax, but for now, let's not worry about it. There are just a few extra lines of wrapping code in class and test methods, as shown in the following code snippet:

```
<?php


class FirstTest extends PHPUnit_Framework_TestCase {

    public function testAddition(){
        $this->assertEquals(2, 1 + 1);
    }
    public function testSubtraction(){
        $this->assertEquals(0.17, (1- 0.83))
    }

    public function testMultiplication(){
        $this->assertEquals(10, 2 * 5);
    }
    public function testDivision(){
        $this->assertTrue(2 == (10 / 5));
    }
}
```

(c) Copyright 2016 OOB

[



Downloading the example code

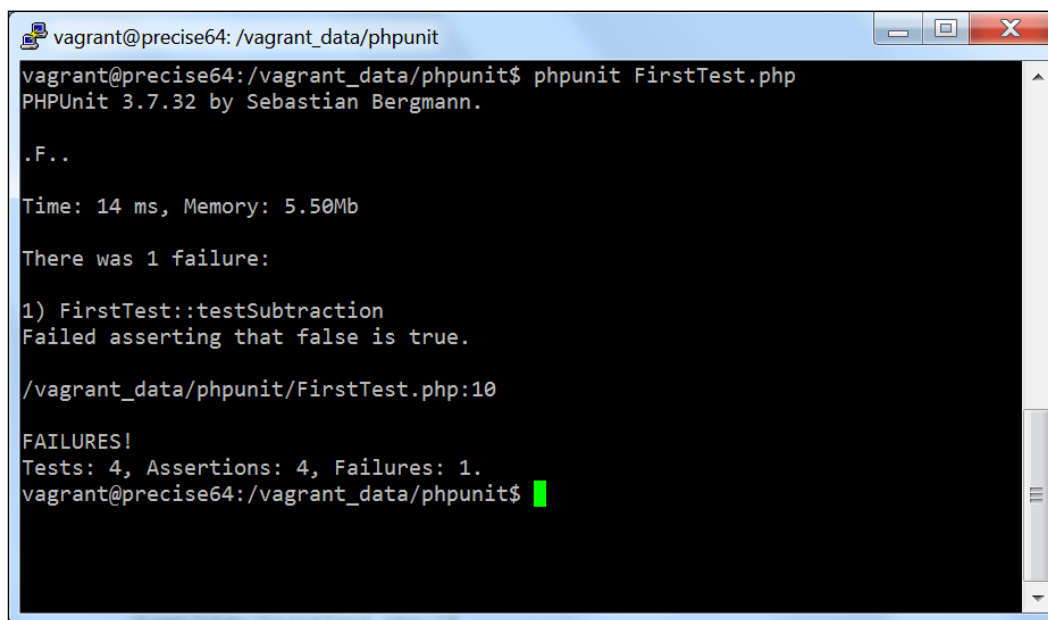
You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

]

Let's save it into `FirstTest.php` and then run it using the following command line:

```
> phpunit FirstTest.php
```

And then we see the result as shown in the following screenshot:



```
vagrant@precise64: /vagrant_data/phpunit
vagrant@precise64:/vagrant_data/phpunit$ phpunit FirstTest.php
PHPUnit 3.7.32 by Sebastian Bergmann.

.F..

Time: 14 ms, Memory: 5.50Mb

There was 1 failure:

1) FirstTest::testSubtraction
Failed asserting that false is true.

/vagrant_data/phpunit/FirstTest.php:10

FAILURES!
Tests: 4, Assertions: 4, Failures: 1.
vagrant@precise64:/vagrant_data/phpunit$
```

Brilliant, it works!

Works, but it says one failure? And it's correct. This example shows you why it is important to run tests. Even things that seem obvious and appear to have no reason to be tested might not be as straightforward as you think.

Let's change the problematic line 10 with the following line of code:

```
$this->assertEquals(0.17, 1 - 0.83);
```

Now run the test again. The result now should be as shown in the following command line:

```
OK (4 tests, 4 assertions)
```

If you wonder why the previous test failed, it was that type float and floating point numbers have limited precision, so they are not exactly equal. The second number is approximately 0.16999999999999999, but when the assertion method was changed to equals, PHPUnit is able to detect and handle it correctly.

Xdebug

As mentioned earlier, Xdebug is not required, but it is really recommended that for development and for running PHPUnit tests, you should install this brilliant PHP extension created by Derick Rethans. The following are killer features of Xdebug:

- **Remote debugging:** This allows to stop code execution, step line by line through the code execution, watch variables, and much more.
- **Code coverage analysis:** This tells you which lines were executed. This is important for testing to know if all code is covered by tests or how much is missing.
- **Code profiling:** During the code execution, it is measured how much time and how much resource each class, method, and function needs. This is handy when you need to identify bottlenecks in your application.

Another similar option is Zend Debugger that is pre-installed in Zend Server, but Xdebug provides better support for code coverage, profiling, and so on.

Installation is the same as any other PHP extension. If you are not using package manager, the tricky bit could be to figure out which one is the right one for you. On the other hand, if you have installed the web server bundle, it might be already there; check the extension directory or `phpinfo()` output.

Installing Xdebug

If Xdebug is not installed, the following are three installation options available for it:

- Use the precompiled binary version of Xdebug. This version needs to match your PHP version. The <http://xdebug.org/download.php> link contains lots of Windows binaries. It is necessary to have the correct PHP version, the compiler that was used for your installation (VC6, VC9, VC11), and if you have it, the thread-safe installation. You can see all of these things in `phpinfo()` output. On Linux, it is much easier to install if you use your package manager. For example, run the following command line:

```
>apt-get install php5-xdebug
```

Installing PHPUnit

- PECL installation (for instructions on how to install PECL, see the *PEAR installation* section discussed earlier). If you have PECL installed, just run the following command line:
`pecl install xdebug`
- Use compiling extension from the source code to install Xdebug. This method is only recommended for experienced users.

To enable the extension, you have to add the following extra line to the configuration file:

```
zend_extension="/usr/local/php/modules/xdebug.so"
```



The path in the configuration file needs to be an absolute path, not a relative one as it is for other PHP extensions.

To check whether the extension is installed correctly and loaded by PHP, just run the following command line:

```
>php --version
```

If extension is loaded, you should see an output similar to the following screenshot:

```
vagrant@precise64: /vagrant_data/phpunit
vagrant@precise64:/vagrant_data/phpunit$ php --version
PHP 5.3.10-1ubuntu3.10 with Suhosin-Patch (cli) (built: Feb 28 2014 23:14:25)
Copyright (c) 1997-2012 The PHP Group
Zend Engine v2.3.0, Copyright (c) 1998-2012 Zend Technologies
    with Xdebug v2.1.0, Copyright (c) 2002-2010, by Derick Rethans
vagrant@precise64:/vagrant_data/phpunit$
```

Summary

There are several ways to install PHPUnit. This chapter showed all the available options. The Composer and PEAR installation methods are recommended as they can handle all the required dependencies and allow to keep PHPUnit and the required libraries up to date. PHPUnit installation is not difficult if the right path is followed. Xdebug extension is optional, but it is strongly recommended that you install it as we will need extra features such as remote debugging and code coverage later.

When writing and running PHPUnit tests, it's really important to have a good **IDE (Integrated Development Environment)**. All modern IDEs have good PHPUnit support. The next chapter will show the four most popular IDEs in PHP world. We will also see how to configure them, how to execute tests, and how to debug tests. Good support for PHPUnit and PHP in general is something that makes a developer's life much easier, and he/she is able to produce good code much faster than using an ordinary text editor.

To have PHPUnit installed is the first good step. In the next chapter, before jumping to writing tests, it's good to have a look at how to run and debug PHPUnit tests in IDE such as Zend Studio and NetBeans.

2

PHPUnit Support in IDEs

The first chapter was about how to install PHPUnit. The next step before writing tests is to prepare the development environment for testing.

A good IDE helps to write unit tests and integrate them with the code, of course, there are many options. There are advanced editors available such as vi or vim, emacs, TextMate, and Notepad++, but IDE makes the developer's life much easier by assisting when you write the code, helping to debug and highlight errors, and much more. As a developer, it's the first thing that you start in the morning and the last thing that you switch off in the evening. A good IDE is like a Swiss army knife, offering everything that is needed in one application. Of course, it's every developer's choice, but this chapter shows what can be done with the four most popular IDEs in the PHP world when writing, running, and debugging tests.

Different IDEs offer different features, but you always need an IDE to run and debug PHPUnit tests. This chapter describes PHPUnit support in two open source products and two commercial, but exceptional, products. All products are Java applications, which means they are cross-platform applications working on Windows, Mac OS X, and Linux. These IDEs have very good integration support for PHPUnit, which is not just for launching PHPUnit as an external tool. This is important when you want to use remote debugging to step through the code, and be able to see exactly what's going on when tests are failing and you are not sure why. Even if you are using different IDEs, a good one has very similar features but maybe in different places.

We chose the following four IDEs mainly because they provide the best features or are the most used:

- NetBeans
- Zend Studio
- Eclipse PDT
- PhpStorm

The following sections not only describe how each of the IDEs support PHPUnit, how to run tests from IDE and get results, but also how to use remote debugging and step through the code execution. Debugging in IDE is a very useful thing, but many PHP developers still use only `var_dump()` and `die()` as their main debugging techniques. It is really bad to commit them to the repository and release to production servers. It not only looks really bad and possibly will break an application, but it might also be a serious security risk when exposing application internals.



This chapter describes IDE's integration with PHPUnit Version 3.7. PHPUnit Version 4 is distributed as a PHAR file (PHP archive), and at the time of writing this chapter, it was not supported by IDEs. However, as soon as updated versions are available, functionality and configuration should be very similar to the following section.

IDEs and PHPUnit

What is required is to have an already installed web server with PHP support, PHP-CLI and PHPUnit, even if some IDE's might have built-in PHP or PHPUnit support.

In all IDEs, PHPUnit support is very similar, and to be fair, it should be done the same way. You write a test, which is PHP code extending `PHPUnit_Framework_TestCase`. IDEs help to validate code, and show all assertions that can be used and their parameters by parsing PHPDoc comments in the code. A good developer writes comments, and in PHP, which is a weakly typed language, you can use it as a helper to notify the IDE about object types when it's not able to detect them from input parameters.

Before looking at how to configure and run tests, it would be good to mention the following two files that are used when executing tests:

- `bootstrap.php`: This is the file that is run before tests. This file is usually used for autoloading classes, so it's not necessary to run `require_once` for every file that needs to be loaded.
- `phpunit.xml`: This is the XML configuration file for tests.

These files are going to be described later in this book; for now it is enough to know what they are because they are mentioned in some of the configuration dialogs.

When PHPUnit is installed, its classes can be used in our project. Before being able to utilize PHPUnit support, usually a bit of configuration is necessary. It doesn't matter which IDE is used, but the following steps might be required:

1. Set which PHP interpreter is used, that is, point it where PHP binary is stored.
2. Set the path to PHPUnit script.
3. Set the include path to PHPUnit classes.
4. Set which debugger is used, for example, Xdebug on standard port 9000.

To be able to run PHPUnit tests, usually, IDE needs to know which PHP interpreter is used, where PHP binary is stored, where PHPUnit classes are, and possibly where PHPUnit script is stored. This is good to know because then you can tweak the configuration to be able to choose any environment/version you need for each project. Another option is to use IDE that has built-in support for everything, such as Zend Studio. The advantage is that you can immediately start writing and running tests without any configuration changes, but the disadvantage is that you haven't got full control over which version is used. Usually, it is good to have matching configuration for the development environment and production servers.

NetBeans

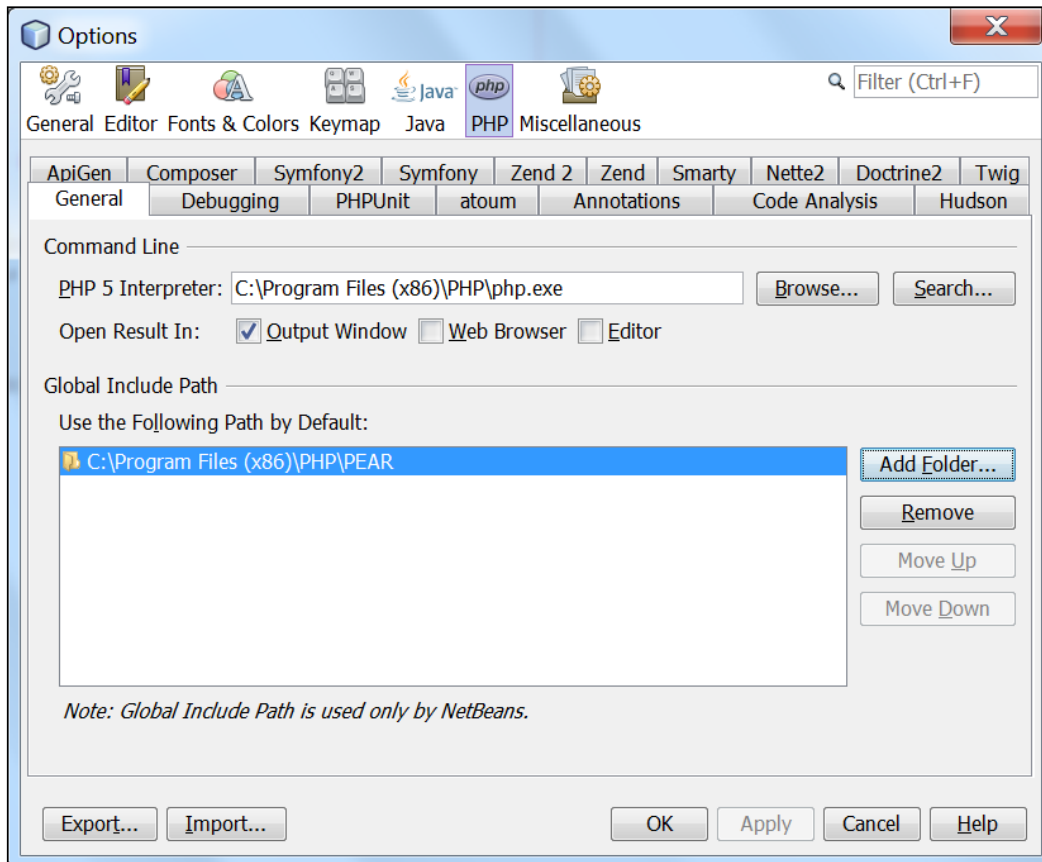
NetBeans is a very well known IDE in the Java world. It was bought by Sun Microsystems and released as open source and is now supported by Oracle. A few years ago, this proved to be an added support for PHP, and NetBeans became a good IDE for PHP developers. It is popular among beginners because the user interface is simpler to use than Eclipse-based IDEs and also more user friendly. NetBeans can be downloaded from <https://netbeans.org>.

To test what can be done with the PHPUnit, let's create a project and use tests that we used for validating installation by performing the following steps:

1. As a first step, create a new project as a PHP application by navigating to the **File | New Project** menu.

PHPUnit Support in IDEs

- Then, for configuration, set the path to the PHP interpreter and PHPUnit. Navigate to **Tools | Options** and there click on the **PHP** tab. The following screenshot shows the **Options** window:



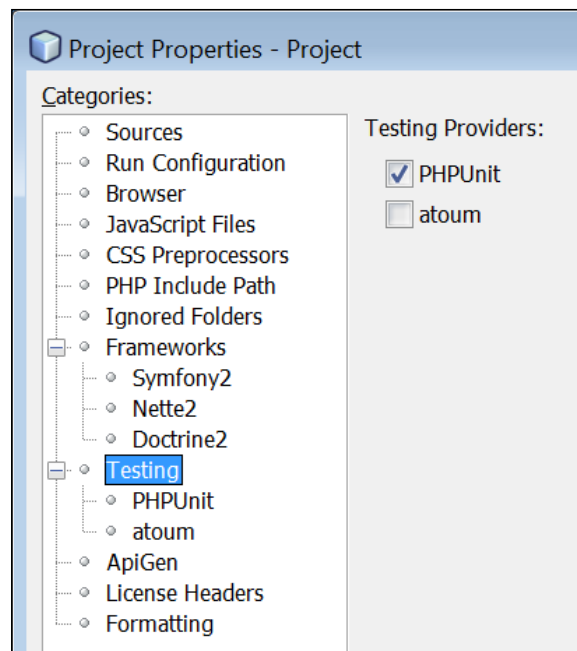
Here, PHP binary needs to be set in the **PHP 5 Interpreter** field.

- Then, add PHPUnit path to **Global Include Path** as you can see in the preceding screenshot. Of course, depending on your operating system and PHP installation, the location is going to be different. We are doing this because we are extending `PHPUnit_Framework_TestCase`, and we need to inform the IDE where to look for PHPUnit classes.

4. The last thing that we need to change is on **PHPUnit** under the **Testing** section; set the path to the PHPUnit script.

To simplify things, there is going to be just the `FirstTest.php` file in the project. We can change project properties in the PHPUnit section under **Testing** by clicking **Run All *Test Files using PHPUnit**, all files with filename ending with `*Test.php` will be executed by PHPUnit.

5. In the latest version of NetBeans, you have to enable **PHPUnit** as **Testing Providers** as shown in the following screenshot:

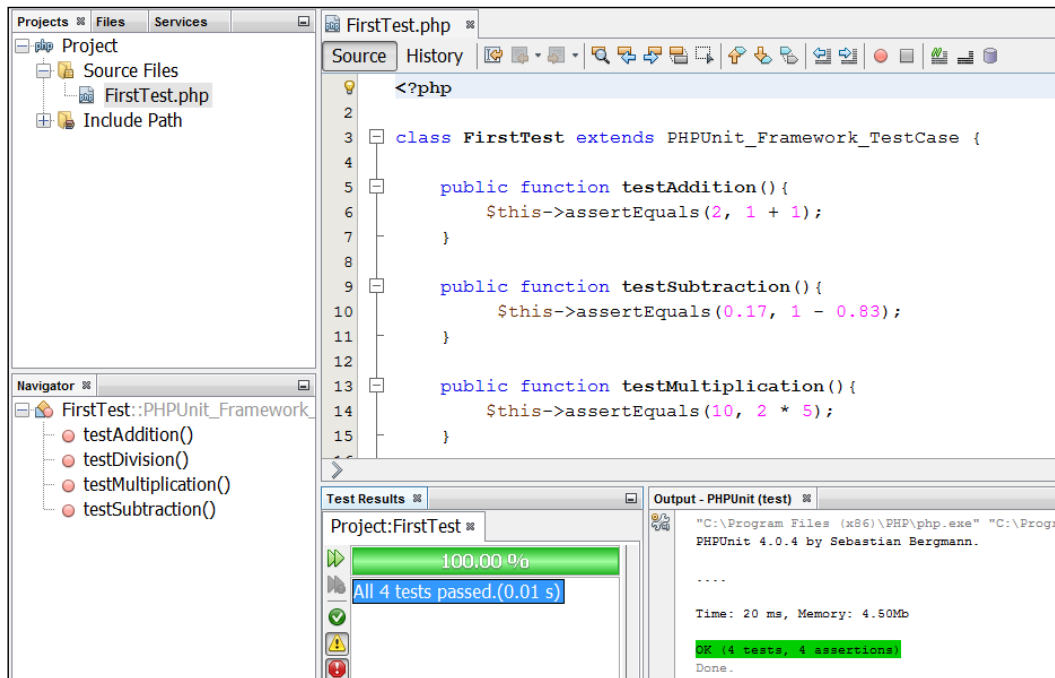


After configuring IDE, it's time to create the first test suite. Create a new PHP class and name it `FirstTest`.

6. When the file is created, copy the source code for `FirstTest` from *Chapter 1, Installing PHPUnit*, and save the file. To run tests, just click on **Run file** or press `Shift + F6`. Since all test files end with `Test.php`, NetBeans detects its PHPUnit test and runs it as a unit test.

PHPUnit Support in IDEs

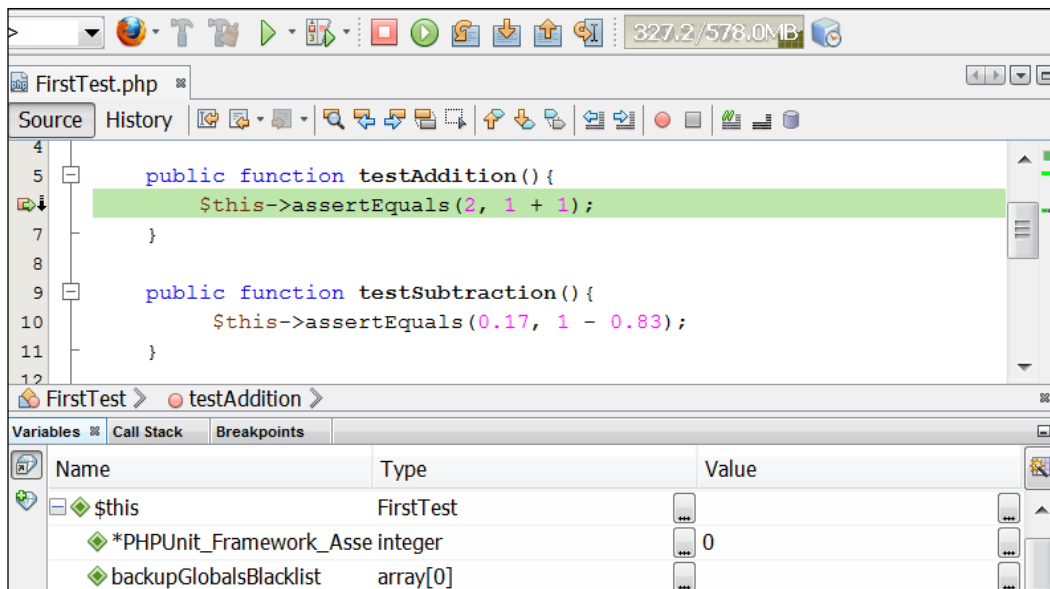
Then you should see similar output saying that all tests are passed as shown in the following screenshot:



7. To debug tests, just put the breakpoint in the place where you want to stop execution by clicking on the row number.

Debugging can be done by clicking on **Debug** or pressing *Ctrl + Shift + F5*. The code is executed, and then it stops on the breakpoint.

In the toolbar, now you can see extra buttons: the debug toolbar for stepping over, stepping in, or stopping debugging. Under the code windows, you have a couple of extra windows for seeing **Variables**, **Call Stack**, and **Breakpoints**, as shown in the following screenshot:



NetBeans is really easy to configure and use, and it gives you very nice and smooth PHPUnit support.

Zend Studio

Zend Studio is a commercial product from Zend. It is a very mature, advanced IDE-based on Eclipse, which is a platform developed by IBM, originally as IDE for Java development, but later adopted as a platform for IDEs for many programming languages.

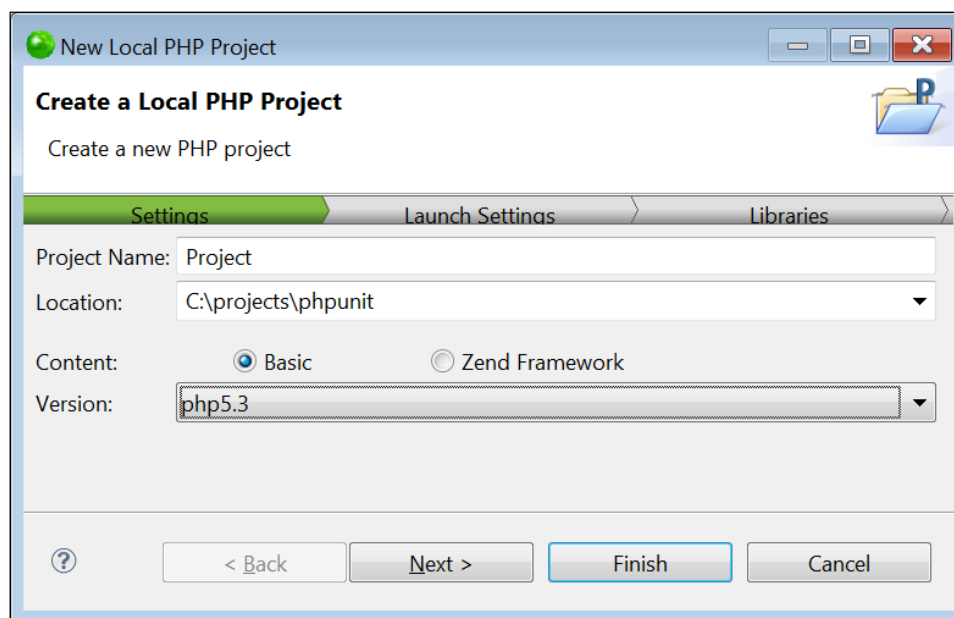
Eclipse is a heavyweight champion, so it might take a little while to get used to it, but then you will see how advanced and flexible a platform it is.

Zend Studio can be downloaded from <http://www.zend.com/en/products/studio>.


PHPUnit Support in IDEs

To see Zend Studio PHPUnit support, let's use `FirstTest` from *Chapter 1, Installing PHPUnit*. Zend Studio configuration is very easy and straightforward as shown in the following steps:

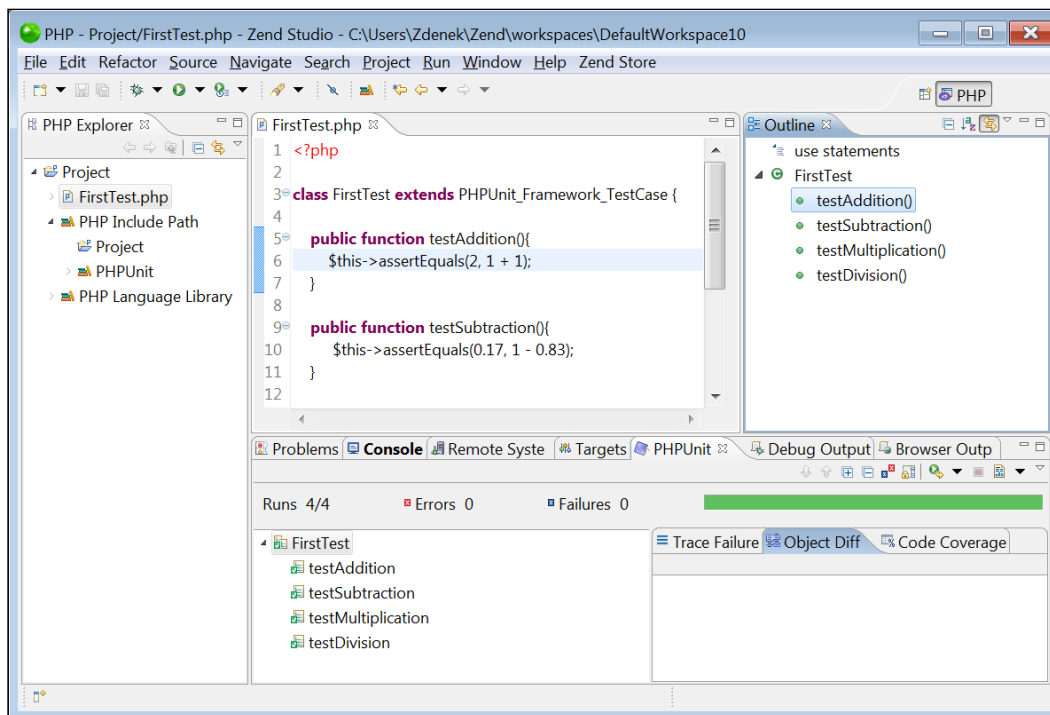
1. As a first step, we need to create a new project. Navigate to **File | New** and select **Local PHP Project**, as shown in the following screenshot:



2. Click on **Next** and select **Launch CLI Application**, and then click on the **Next** button.
3. In the **Libraries** tab, select the **PHPUnit [built-in]** library option. Then, click on the **Finish** button. This will create a new project. Now, add PHPUnit to **PHP Include Path**.


 PHPUnit in Zend Studio is not your installed PHPUnit, but the one shipped with Zend Studio. If you want to change it, go to project properties, remove this library, and add a location where your PHPUnit is stored.

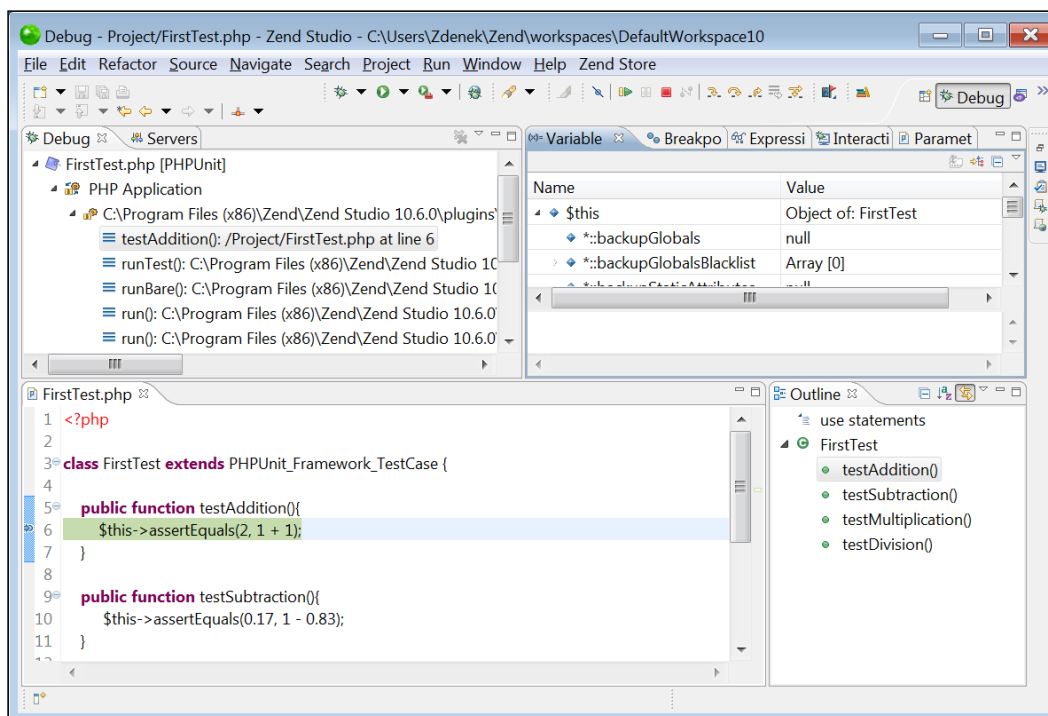
4. To create our test case, right click on the **Project**, navigate to **New**, and select **PHPUnit Test Case**.
5. Replace the content of the generated class with that of our `FirstTest.php`. Then to run test, just right-click in the code, and select **Run As** and then **PHPUnit Test** or press *Alt + Shift + X*, and then select **PHPUnit test**. Then you should see a similar result as shown in the following screenshot:



6. To be able to debug tests, put the breakpoint next to line number by double clicking there.
7. To debug the test, right-click anywhere in the code and select **Debug As** and then **PHPUnit Test** or press *Alt + Shift + D*, and then select **PHPUnit test**.

PHPUnit Support in IDEs

- Then, confirm switch to the debug perspective, and execution is stopped on the breakpoint. Now you can go step by step through code execution as shown in the following screenshot:




Zend Studio comes with everything that you need. It is even shifted with its own bundled PHP and the PHPUnit version so that you don't have to worry about the configuration too much. But if you want to use your own libraries and configuration, then you have to do some configuration changes. PHPUnit support is very good and easy to use.

Eclipse PDT

PDT means PHP Development Tools. It is an open source version of Zend Studio and can be downloaded as an all-in-one package from <http://www.zend.com/en/company/community/pdt/downloads> or installed as a plugin to Eclipse. As compared to Zend Studio, it has lesser features, but by using Eclipse plugins, you can add most of the functionality, which Zend Studio has built in. One of the features is support for PHPUnit. A brilliant Eclipse plugin called **MakeGood** is available.

You can find out more about this project on the project wiki page at <http://piece-framework.com/projects/makegood/wiki>.

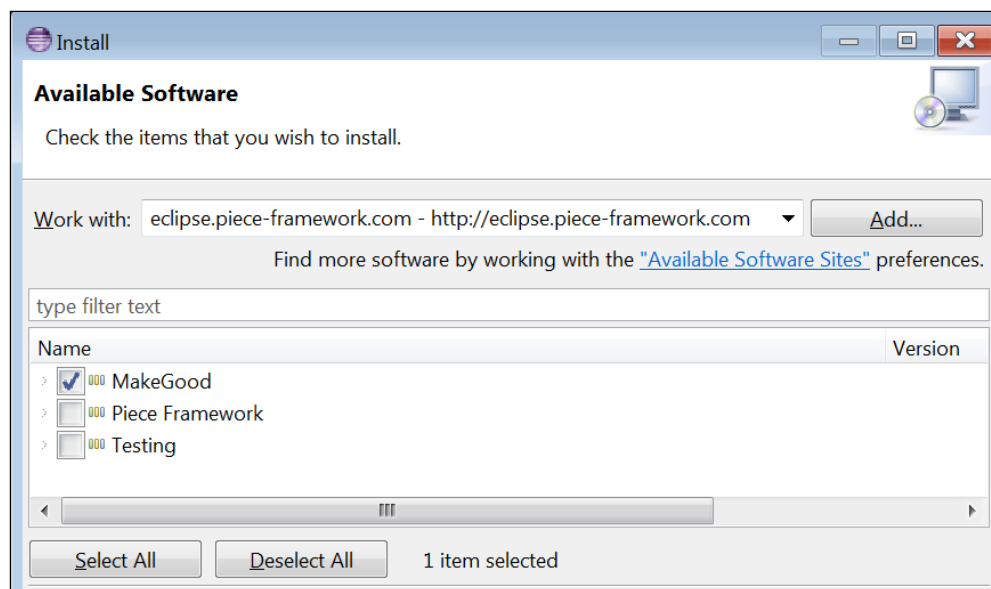
 There is one interesting thing about Eclipse that hasn't been mentioned earlier in this chapter. Besides being IDE for many languages and platforms, Google chose Eclipse as the official platform for Android application development through **Android Development Tools (ADT)**. So, if you tried building something for Android or you are planning to, knowing Eclipse really helps.

When you have Eclipse PDT installed, a few configuration tweaks are needed. You also need to install MakeGood because it's not installed by default.

Installing MakeGood

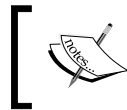
Perform the following steps to install MakeGood:

1. Navigate to **Help | Install New Software...**, and then click on **Add...** to add from where to download the plugin (<http://eclipse.piece-framework.com>), as shown in the following screenshot:



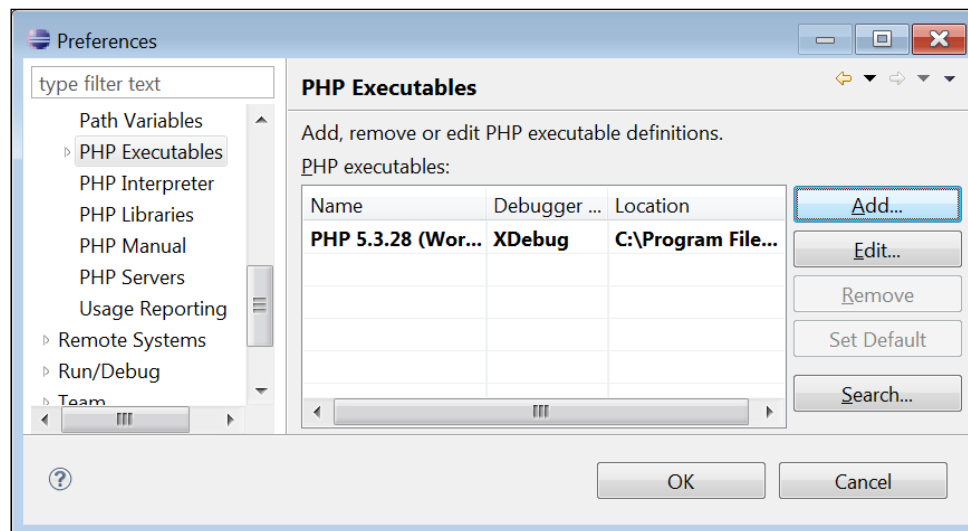
PHPUnit Support in IDEs

2. Select **MakeGood**, and then click on **Next**. Go through the installation wizard and install the plugin. After successful installation, you will be asked to restart Eclipse. If you are having problems with installation, try cleaning PDT's all-in-one installation, and before installing the plugin, check for updates and install all available updates.

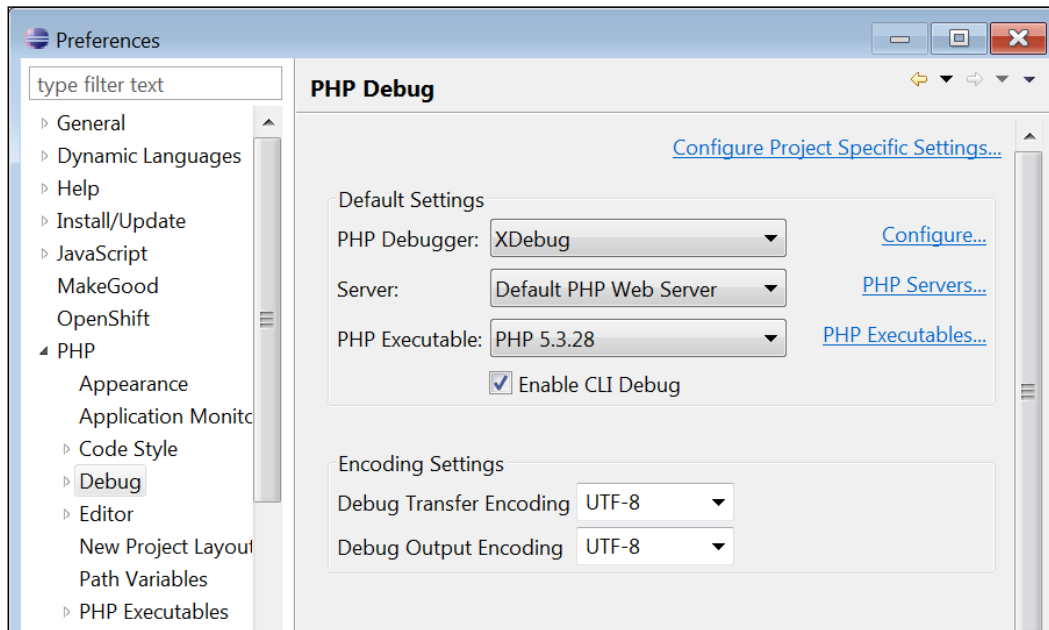


After installing PDT with brilliant support for PHPUnit, the same way you can find and install additional plugins such as support for GIS or SVN.

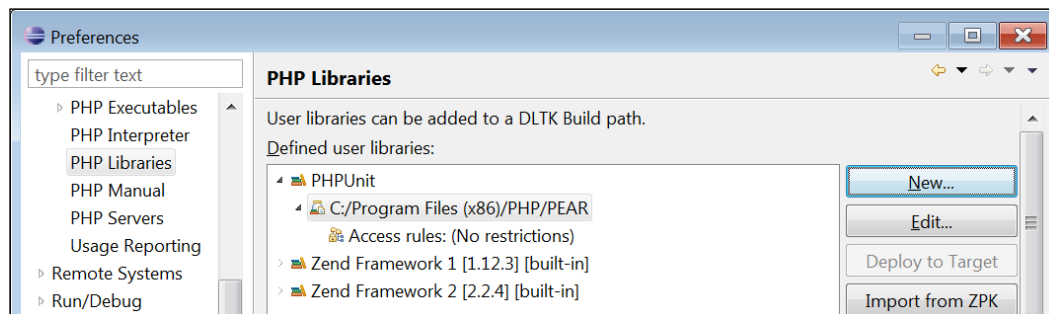
3. Now a few configuration changes need to be done. Go to **Window** and select **Preferences**.
4. Under the **PHP** section, select **PHP Executables**, and point it to the location where your PHP binary is stored. It is safer to manually point it to the location where PHPUnit is stored, instead of using the search functionality. Also, switch the debugger option to **Xdebug** as shown in the following screenshot:



- Now, select the **Debug** option. In the **PHP Debug** section, set **PHP Debugger** to **Xdebug** and select your configured PHP executable. Also, check **Enable CLI Debug**, as shown in the following screenshot:

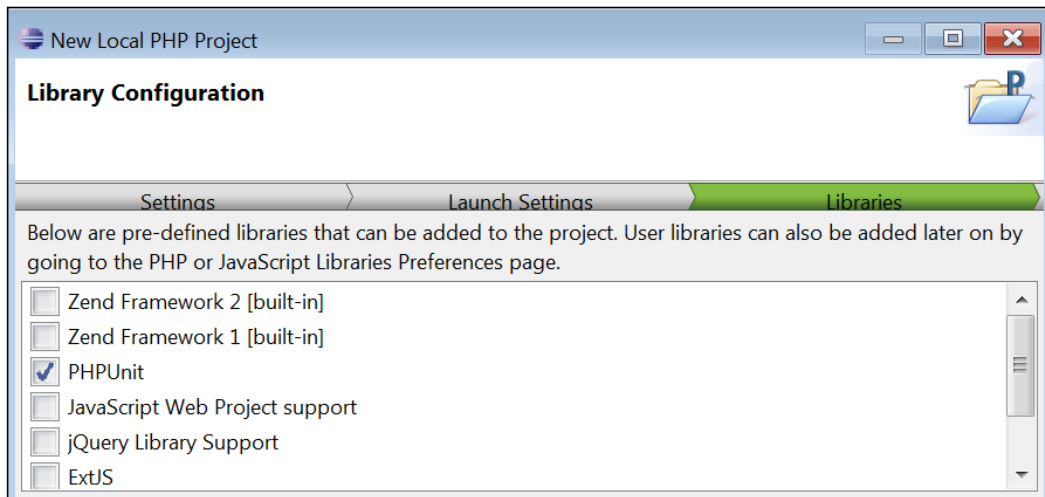


- Now add **PHPUnit** to **PHP Libraries**, that is, two options below **PHP Executables**.
- Click on **New**, and then add an external folder. Point to the location where the PHPUnit classes are stored, as shown in the following screenshot:

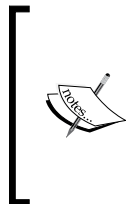


PHPUnit Support in IDEs

8. Now configuration is done, and you can create a project. Go to **File** and select **New project**. The project needs to be of the type Local PHP project. If you can't see it in the menu, click on **Other** and scroll to the **PHP** section.
9. Click on **Next** and select **Launch CLI Application**, and then click on the **Next** button. In the **Libraries** tab, select the **PHPUnit** library option, as shown in the following screenshot:



10. Now click on **Finish** and switch to the PHP perspective.



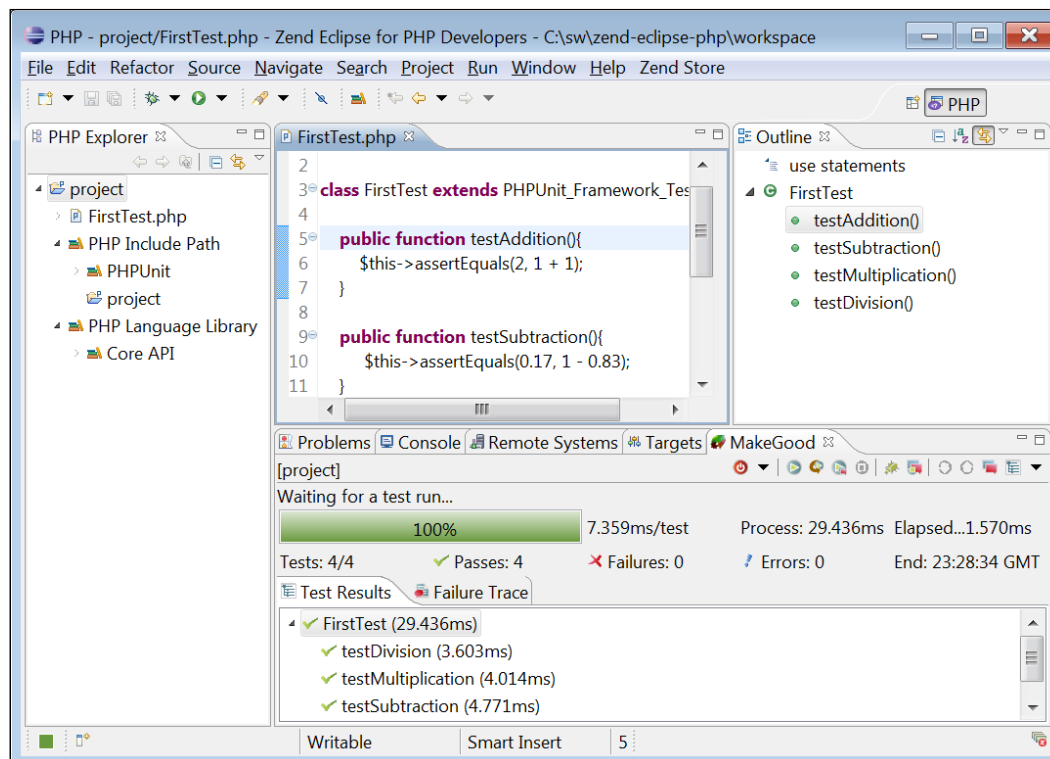
Perspectives and views are very important things in Eclipse. In every window, view can be fully customized. This can sometimes cause problems for beginners; it might be quite easy to get lost. For PHP, there are two most important perspectives: PHP for writing code and PHP debug for debugging code.

PDT is now configured to have a full built-in support for PHPUnit.

Creating your FirstTest

For creating your `FirstTest.php`, perform the following steps:

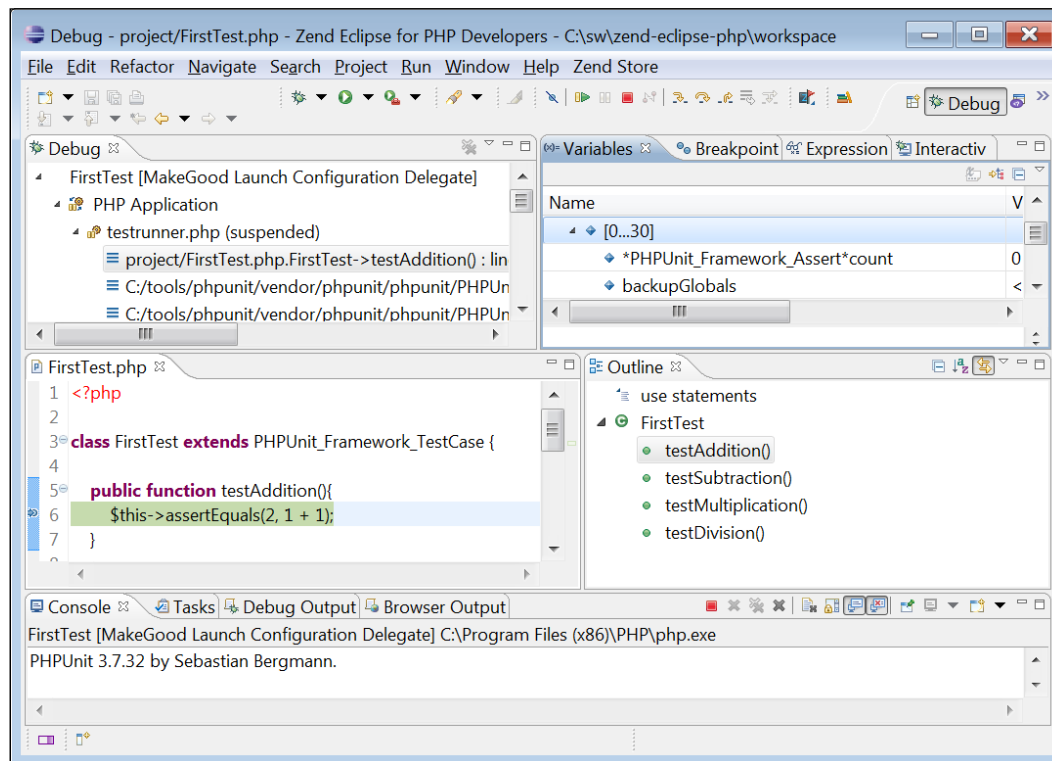
1. To create your `FirstTest.php` file, right-click on project and select **New PHP file**.
2. In the newly created file, copy the code from `FirstTest`, which we used in *Chapter 1, Installing PHPUnit*. To run all tests, right-click anywhere in the code and select **Run Tests In File** or **Class**, or press `Alt + M` to see run options. After selecting **Run Tests In Class**, you should see a similar output to that shown in the following screenshot:



3. There might be something different on your screen. If you can't see the **MakeGood** tab, navigate to **Window | Show View | Other** and select **MakeGood**.

PHPUnit Support in IDEs

4. Debugging is very easy now. Double-click in the column next to row number to toggle breakpoints. To switch **MakeGood** to debug mode, click on the bug icon in the **MakeGood** view.
5. When you run the test again, you will be asked to switch to the debug perspective. Execution is going to be stopped where you put a breakpoint, and you can step through the execution line by line, as shown in the following screenshot:



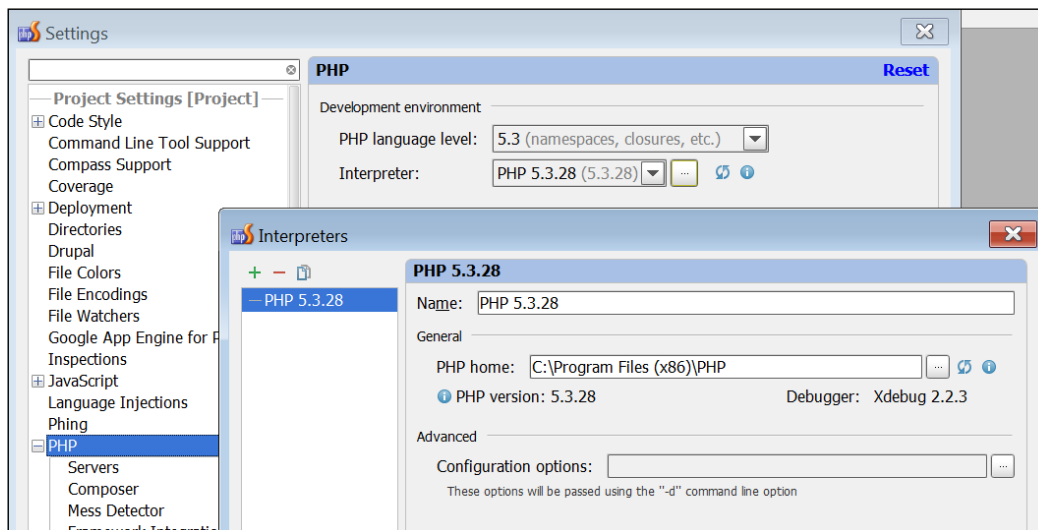
Eclipse PDT with MakeGood is a very good IDE. But you have to do a few configuration tweaks before you can use it. An advantage is that you can fully customize it to your needs and configuration. Then it's a very interesting and free alternative.

PhpStorm

PhpStorm is an advanced IDE based on IntelliJ, a very popular IDE in the Java world. It has support for almost everything that you need for web development, but more importantly, very good support for PHP and PHPUnit. It is similar to Zend Studio in its features, and seems to be faster than Eclipse-based products.

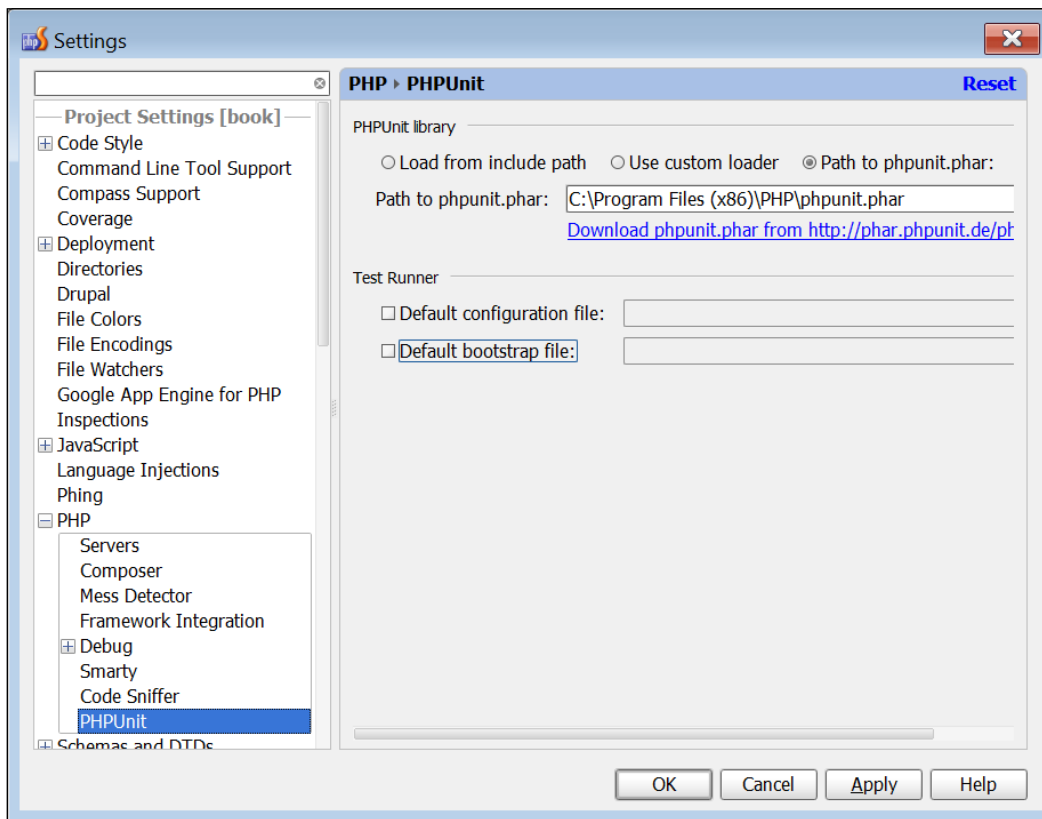
PhpStorm is a commercial application and can be downloaded from <http://www.jetbrains.com/phpstorm>. PHPUnit configuration in PhpStorm is not difficult. Perform the following steps for installation:

1. As a first step, you need to create a new project. Go to **File** and select **New Project...**
2. Now in this project, you need to configure where the PHP interpreter is installed. Go to **File** and select **Settings...**, and there select **PHP**. Now you need to choose which version of PHP you are using and where PHP binary files are stored. Also, you should have installed Xdebug, so select **Xdebug** in the **Debugger** field, as shown in the following screenshot:

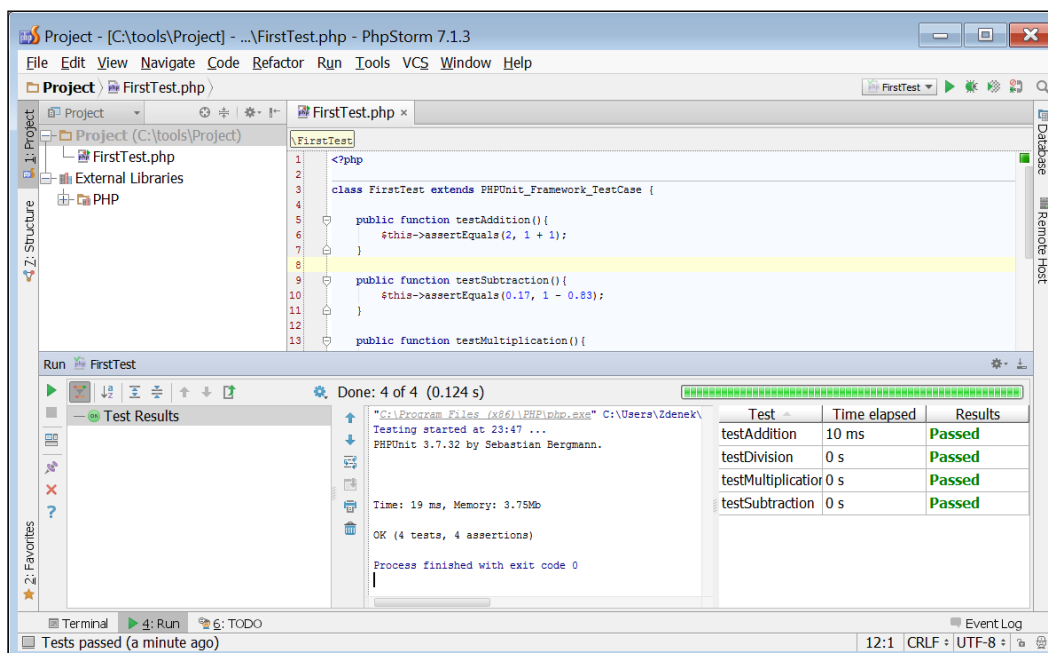


PHPUnit Support in IDEs

- The next thing that we have to do is set the path to the location where PHPUnit is installed. Navigate to **File | Settings | PHP | PHPUnit**, and here you have multiple options. For PHPUnit 3.7, it is best to select **Load from include path** and point it to where PHPUnit is installed, for example, `C:\Program Files (x86)\PHP\PEAR`. For PHPUnit 4, you will need PHPStorm 8 or a higher version; select **Path to phpunit.phar**, as shown in the following screenshot:

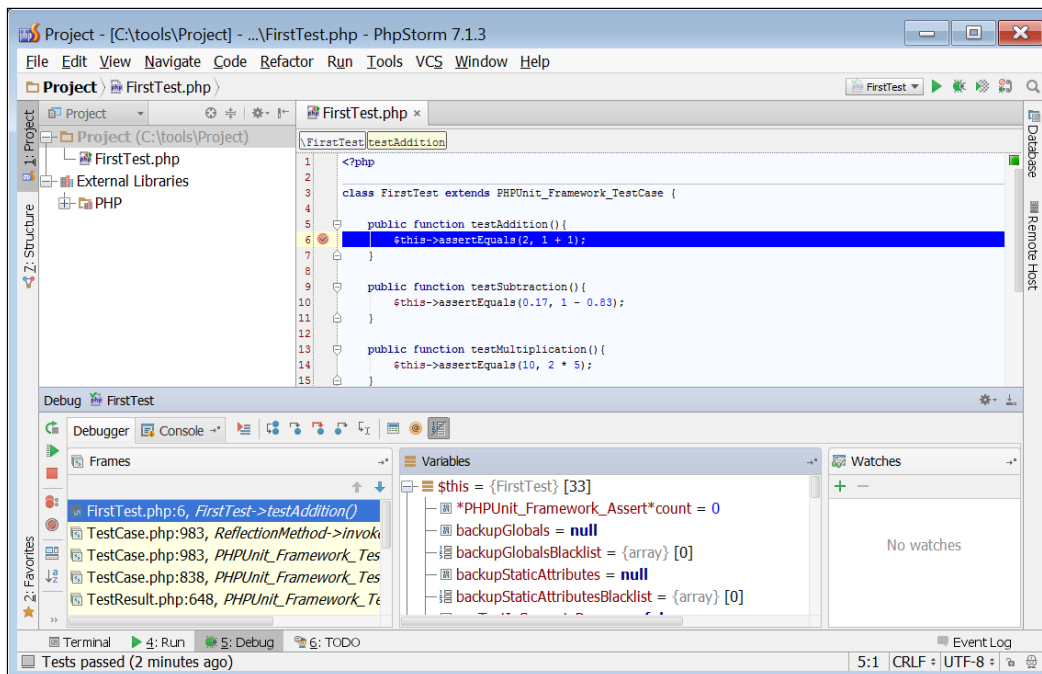


4. Now PhpStorm is configured to be able to write and run tests. Let's use our first test. To create file and test class, right-click on your project and navigate to **New | PHPUnit | PHPUnit Test**. Give it the name `FirstTest` and namespace `Tests`.
5. Now, you will have an empty class extending `PHPUnit_Framework_TestCase`. Let's copy the test methods that were used for testing installation in the previous chapter. Then, if you right-click anywhere in the code, or you go to **Run** item in the menu, or if you press **Alt + Shift + F10**, you run all tests in the test class and should see an output similar to the following screenshot:



PHPUnit Support in IDEs

- Now if you click between line number and code, you can put the breakpoint there. When you run test in debug mode, execution will be stopped. IDE brings debug view, where you can see all the variables; you can step through the code, and see code execution step by step. You start debugging the same way as running tests; just select **Debug** instead of **Run**. If you run debug and the breakpoint is in the right place, you should see a result similar to the following screenshot:



PhpStorm requires a bit of configuration to be able to run the PHPUnit tests. But otherwise, it's a brilliant IDE, and to run and debug tests is really easy.

Summary

To be able to use PHPUnit effectively, it is important to have good tools available. Most advanced IDEs have good PHPUnit support. Configuration could be a bit tricky, but it's an investment that pays off later. Good IDE must support PHPUnit test execution and debugging as basic, must-have features. Many of them support more features such as generating and displaying code coverage, but the developer always needs the basic features mentioned in this chapter and advanced features for later use.

In this chapter, we described four IDEs, but any other good IDE will have similar configuration and PHPUnit support, and it is up to the developer to choose the one that suits him/her best. After the two introductory chapters, the next chapter will explain what are unit tests and when and why to write them.

3

Tests and What They're All About

In the first two introductory chapters, we installed PHPUnit and set up IDE to be able to write and run tests. Now it's time to jump into the unit testing swimming pool and swim. Lots of theory has been written about unit testing, test-driven development, behavior-driven development, and many more clever things. But the developer needs to be able to use technology as quickly as possible and see results, and this is what we try. You will see step by step how you can use it: how to write code that can be tested and code that can't be tested. As PHP is the language used by web developers, the focus of unit testing is going to be web applications. It's important to understand the structure of the application or framework that you are using. Good design patterns especially MVC are something that really help to write robust and maintainable code and also help a lot when you are writing tests. Simply put, testable code is very often better code.

In this chapter, we are going to look at the following topics that not only include basic PHPUnit tests, but also writing and running tests in the web application context:

- Understanding unit testing
- The anatomy of a PHPUnit test
- Testing functions
- Testing methods
- MVC application architecture and tests
- Testing controllers

Understanding unit testing

Before looking at code, it would be good to mention some basic information and rules for writing unit tests and using unit testing. It's really important to remember these few basic rules and understand the point of unit testing. Unit tests are not just a nice feature, but are an absolutely necessary part of any decent software project.

What is a unit test?

A simple question is what is a unit test? A unit test is code that executes another piece of code (function/method) in a known context with known inputs, and compares the output to the expected values. This is also called an assertion. The following code snippet is the simplest assertion verifying that one plus one equals two and if function sum works:

```
function sum($a, $b)
{
    return $a + $b;
}

$this->assertEquals(2, sum(1, 1));
```

Assertions

An assertion is the heart and soul of unit testing. An assertion comes together with constraint. Your `assertThat` value suits constraint. A good example of how this works is the simplest assertion available in PHPUnit `assertTrue()`, as shown in the following code snippet:

```
public static function assertTrue($condition, $message = '')
{
    self::assertThat($condition, self::isTrue(), $message);
}
```

The following are basic and the most used assertions:

- `assertTrue()`: This verifies that a condition is true
- `assertFalse()`: This verifies that a condition is false
- `assertEquals()`: This verifies that expected and actual values are equal, the same way as the PHP comparison operator `==`
- `assertSame()`: This is similar to `assertEquals()`, but it checks whether values are identical, the same way as the `===` operator

- `assertNull()`: This verifies that value is null
- `assertEmpty()`: This verifies that value is empty, but it uses the PHP function `empty()`, which means empty can be false, null, '', array()

But PHPUnit has many different built-in assertions. You can find a complete list of these assertions in official documentation at <http://phpunit.de/manual/3.7/en/appendixes.assertions.html>.

But if you need more, you can create your own assertion by extending the `PHPUnit_Framework_Constraint` class.

The importance of unit testing

Unit tests give us confidence that written code works as expected and it's a solid piece on which a developer can rely. Breaking code into small independent units reduces the risk of introducing bugs when the code interacts with another piece of code. It's the best investment available if you need to expend and modify an application without worrying about any unexpected consequences of your changes.

It can also be a very good piece of documentation to describe how code was designed and what it's supposed to do. Another reason is refactoring. Changing complex code without testing is like stepping into the mine field.

Testing all possible scenarios

It would be nice to test all possible scenarios but consider a function as shown in the following code snippet:

```
function plusOne($a)
{
    return 1 + $a;
}
```

PHP is a loosely typed language, and you can have many scenarios. But it is usually enough to cover the most expected or the most important scenarios and also a couple of unexpected ones (in our case, for example, `$a` can be `NULL` or `FALSE`).

But what you should try is to test as much code as possible, hitting all if else or case statements. Try to assume the worst by testing edge cases. It is important to test not only positive cases, getting expected results, but also negative cases to verify that code wouldn't break when unexpected inputs or exception is thrown.

What makes a good test?

The following are some generic rules that are valid for any unit test and not just PHPUnit:

- **Independent:** Each test needs to run independently from other tests and environments.
- **Fast:** To have useful tests and be able to run them as often as possible (for example, as pre- or post-commit hooks), tests need to be fast.
- **Repeatable:** You should be able to run a test as many times as you want with the same result.
- **Up to date:** Tests are written once, but code can be changed or extended. If you are not going to keep tests up to date, the initial investment in tests will be just a waste of time and money. The rule is, whoever breaks the test, fixes the test.
- **Short:** Tests should be just a few lines — easy to read and understand.
- **Resilient:** Once written, tests shouldn't change till the behavior of tested class/method changes.

When to write tests

It would be nice to have 100 percent code coverage (the amount of code tested, but not necessarily 100 percent code coverage also means all possibilities tested). Usually, better coverage means a better quality project. Check several open source projects on GitHub to see how many tests they have, and this gives you a clue about the quality of these projects, and it doesn't matter how often these projects are used. A widespread system doesn't automatically mean high quality code.

But what is more important is to get into the habit of writing tests. You can try the test-driven development approach, where you can imagine your class like an interface with no implementation, just saying how each method should work. You then write tests describing exactly what you expect, and then you implement the interface — writing the required functionality and the test will verify that it does what you expect.

A second way is to write the class or function and then test it with written tests to verify functionality. The rule should be that you write the test the same day as you wrote the code, because later you are not going to do it as you will be focusing on something else.

It is absolutely necessary to have the entire core functionality covered by unit tests. This is a must-have feature.

Anatomy of a PHPUnit test

Now, let's have a closer look at what the test structure looks like. Let's start with a simple test case, which will show the basic PHPUnit test structure. The following code snippet is a very basic example testing two PHP functions used for sorting arrays: `asort()` is used to sort an array and maintaining the indexes, and `ksort()` is used to sort the array by key. To start with, we have an array of vegetables, where name is the key and price is the value:

```
<?php

class SecondTest extends PHPUnit_Framework_TestCase
{
    public function testAsort()
    {
        $vegetablesArray = array('carrot' => 1, 'broccoli' => 2.99,
        'garlic' => 3.98, 'swede' => 1.75);
        $sortedArray = array('carrot' => 1, 'swede' => 1.75,
        'broccoli' => 2.99, 'garlic' => 3.98);
        asort($vegetablesArray, SORT_NUMERIC);
        $this->assertSame($sortedArray, $vegetablesArray);
    }

    public function testKsort()
    {
        $fruitsArray = array('oranges' => 1.75, 'apples' => 2.05,
        'bananas' => 0.68, 'pear' => 2.75);
        $sortedArray = array('apples' => 2.05, 'bananas' => 0.68,
        'oranges' => 1.75, 'pear' => 2.75);
        ksort($fruitsArray, SORT_STRING);
        $this->assertSame($sortedArray, $fruitsArray);
    }
}
```



You might be tempted to put in the code `print` statement or use `var_dump()`. This might also be the moment to write the test, to understand what the code is doing, and verify the functionality without needing to open a browser page. Tests are a good way of learning about your code.

Tests and What They're All About

As you can see, every test case is a class. The class name should end with `Test`. This is because when you test your classes, the test class should mirror tested class, and the difference between the tested class and the test suite is `Test` added to the name:

```
.
|-- src
|   |-- library
|       |-- Util
|           |-- File.php
|-- tests
|   |-- library
|       |-- Util
|           |-- FileTest.php
```

Every test class also called test suite extends the `PHPUnit_Framework_TestCase` class. Of course, you can write your own modified or extended `PHPUnit_Framework_TestCase` and call it, for example, `MyTestCase`, but again this parent class should extend the original `PHPUnit` class. This can be seen in the following class definition:

```
abstract class PHPUnit_Framework_TestCase extends
    PHPUnit_Framework_Assert implements PHPUnit_Framework_Test,
    PHPUnit_Framework_SelfDescribing
```

And the following tells you what the preceding class definition does:

- `abstract`: This is used because you always extend the `PHPUnit_Framework_TestCase` class and do not execute this class directly
- `extends PHPUnit_Framework_Assert`: This is the class providing a set of assert methods such as `assertTrue()` and `assertEquals()`
- `implements PHPUnit_Framework_Test`: This is a simple interface containing just one method, `run()`, to execute the test
- `implements PHPUnit_Framework_SelfDescribing`: This is another simple interface containing just one method `toString()`

`PHPUnit` doesn't use namespaces, which have been introduced to PHP 5.3. Mainly for backwards compatibility, `PHPUnit` uses underscores in class names to match the class location in a filesystem.

Defining test methods

Each test method name should start with `test`. Then the name is up to you. You can mirror their testing class methods, but you will probably need more tests. Each test method should have an explanatory name, meaning you should be able to say what you are testing just from the name, for example, `testChangePassword()` or `testChangePasswordForLockedAccounts()`.

Testing functions

That was a bit of theory, now let's have a look at a specific real-life example. Let's use the following question that developers can get during a job interview, and with this example see how PHPUnit can help us to solve the problem or make us confident that we have found right solution.

The task is to write a function in PHP that returns the largest sum of contiguous integers in an ordered array.

For example, if the input is `[0, 1, 2, 3, 6, 7, 8, 9, 11, 12, 14]` the largest sum is $6 + 7 + 8 + 9 = 30$.

As you can see, it doesn't look particularly difficult. You can take several different approaches. Write something really clever, use brute force and a traverse array, but anything that works is the right answer.

Now, let's have a look at how PHPUnit can help to solve this problem. Firstly, you have to stop and think about what you are doing — not just start writing code, and after a while becoming lost, not knowing what you wanted to achieve.

So how do you solve this problem?

The following steps would be the best approach:

1. Use the PHP function `usort()` to sort an array by values using a user-defined comparison function.
2. Then return the last element of the sorted array, which is going to be the largest group.

This is the important bit for figuring out strategy. Now, let's translate it into a function without writing the implementation yet. Consider the following code snippet:

```
/**
 * Returns largest sum of contiguous integers
 * @param array $inputArray
 * @return array
```

(c) Copyright 2016 OOB

Tests and What They're All About

```

*/
function sumFinder(array $inputArray){}
/**
 * Custom array comparison method
 * @param array $a
 * @param array $b
 * @return int
 */
function compareArrays(array $a, array $b){}

```

Now you have two functions. The first function takes array as the input by using the PHP function `usort()`, and second function, `compareArrays()`, sorts the arrays by groups. The first function then returns an array with the group name and sum.

Now you know what the functions will look like. Let's write tests for them that will describe how they will behave, as shown in the following code snippet:

```

<?php

require_once 'SumFinder.php';

class SumFinderTest extends PHPUnit_Framework_TestCase
{
    public function testSumFinder()
    {
        $input = array(0, 1, 2, 3, 6, 7, 8, 9, 11, 12, 14);
        $result = array('group'=>'6, 7, 8, 9', 'sum'=> 30);
        $this->assertEquals($result, sumFinder($input));
    }

    public function testCompareArrays()
    {
        $array1 = array(0,1,2,3);
        $array2 = array(6,7,8,9);

        // $array2 > $array1
        $this->assertEquals(-1,compareArrays($array1,$array2));
        // $array1 < $array2
        $this->assertEquals(1,compareArrays($array2,$array1));
        // $array2 = $array2
        $this->assertEquals(0,compareArrays($array2,$array2));
    }
}

```

As you can see, there are two simple tests for each function. The first test, `testSumFinder()`, verifies the main `sumFinder()` function, and the second test, `testCompareArrays()`, verifies that arrays are going to be sorted correctly when `usort()` is used. Now you can run them and see the following results:

FAILURES!

Tests: 2, Assertions: 2, Failures: 2.

Brilliant! This is what we want: it's failing because we have to write an implementation. Let's start with the `compareArrays()` function as shown in the following code snippet:

```
/**
 * Custom array comparison method
 * @param array $a
 * @param array $b
 * @return int
 */
function compareArrays(array $a, array $b)
{
    $sumA = array_sum($a);
    $sumB = array_sum($b);

    if ($sumA == $sumB) {
        return 0;
    }
    elseif ($sumA > $sumB) {
        return 1;
    }
    else {
        return -1;
    }
}
```

Then you can run `testCompareArrays()`, and great, it works as shown in the following output:

OK (1 test, 3 assertions)

Now we can move on to `sumFinder()` implementation as shown in the following code snippet:

```
/**
 * Returns largest sum of contiguous integers
 * @param array $inputArray
 * @return array
```

Tests and What They're All About

```

    */
function sumFinder(array $inputArray)
{
    $arrayGroups = array();

    foreach ($inputArray as $element) {
        //initial settings
        if (!isset($previousElement)) {
            $previousElement = $element;
            $arrayGroupNumber = 0;
        }

        if (($previousElement + 1) != $element){
            $arrayGroupNumber += 1;
        }

        $arrayGroups[$arrayGroupNumber][] = $element;
        $previousElement = $element;
    }

    usort($arrayGroups, 'compareArrays');
    $highestGroup = array_pop($arrayGroups);

    return(array('group'=> implode(' ',
        $highestGroup), 'sum'=>array_sum($highestGroup)));
}

```

When you run the preceding code snippet, you will get the following result:

OK (1 test, 1 assertion)

And this is the solution. Here you should see one of the biggest advantages of unit testing; it forces you to think about what you are doing and to have a clear idea before you start to write code or clarify it before implementation.

Testing methods

Testing methods are very similar to testing functions, but you can do much more. There is nothing wrong with procedural programming, but there are very good reasons to use **Object Oriented Programming (OOP)**. PHP is strictly not an OOP language such as Java or C#, but this doesn't mean you can't use OOP. Of course you can, and PHP has a very good object model. It's a solid implementation with all the main features you would expect. Where OOP really helps is in organizing your code with all the OOP advantages, and it also helps when you are writing tests.

PHPUnit has strong support for testing classes and objects. The real strength of PHPUnit and unit testing comes when you are testing classes. Not only can you test every method, but when necessary, you can replace part of the class with your implementation just for testing, for example, to avoid storing data in the database. This is called test doubles and these techniques are going to be described later in this book.

When testing class methods, it is recommended to test just public methods. This might sound a bit strange after it was said as much code as possible should be tested. Why not write a test for private or protected methods? The reason is that you should test just public methods; everything else is just internal implementation that you shouldn't worry about, and even if it changes, public methods should still behave in the same way. If private/protected methods are too complex, it might suggest that they deserve their own class, or you can use PHP Reflection and change accessibility on the fly.

As an example of how to test methods, let's use the example that we used for testing functions and adapt it to classes.

So the following task is the same as mentioned earlier.

The task is to write a function in PHP that returns the largest sum of contiguous integers in an ordered array.

For example, if the input is [0, 1, 2, 3, 6, 7, 8, 9, 11, 12, 14] the largest sum is $6 + 7 + 8 + 9 = 30$.

Again, it would be good to start with a class that looks almost like an interface, or if you want, you can create an interface, as shown in the following code snippet:

```
class SumFinderClass
{

    private $inputArray;

    /**
     * @param array $inputArray
     */
    public function __construct($inputArray = null){}

    /**
     * Returns largest sum of contiguous integers
     * @return array
     */
    public function findSum(){}
```


Tests and What They're All About

```

    /**
     * Custom array comparison method to sort by sum of contiguous
    integers
     * @param array $a
     * @param array $b
     * @return int
     */
    public function compareArrays(array $a, array $b){}}

```

The difference with procedural implementation, in this case, is that it just encapsulates all functionality into one class that can be extended and modified if necessary. In this case, you could be tempted to set the `compareArrays()` method as `private`, but in order to test it easily, let's leave it as `public` for now. Now we have a class skeleton, let's write tests as shown in the following code snippet:

```

<?php
require_once 'SumFinderClass.php';

class SumFinderClassTest extends PHPUnit_Framework_TestCase
{
    public function testFindSum()
    {
        $input = array(0, 1, 2, 3, 6, 7, 8, 9, 11, 12, 14);
        $result = array('group'=>'6, 7, 8, 9', 'sum'=> 30);

        $sumFinder = new SumFinderClass($input);
        $this->assertEquals($result, $sumFinder->findSum());
    }

    public function testCompareArrays()
    {
        $array1 = array(0,1,2,3);
        $array2 = array(6,7,8,9);

        $sumFinder = new SumFinderClass();

        // $array2 > $array1
        $this->assertEquals(-1,
            $sumFinder->compareArrays($array1,$array2));
        // $array1 < $array2
        $this->assertEquals(1,
            $sumFinder->compareArrays($array2,$array1));
        // $array2 = $array2
        $this->assertEquals(0, $sumFinder->compareArrays($array2,
            $array2));
    }
}

```

We have two test methods: `testFindSum()` and `testCompareArrays()`, which are testing two public methods in this class. The `testCompareArrays()` method doesn't require any dependency, and we are just passing test data there to verify that the compare method works as expected. The `testFindSum()` method does a full check, of course, internally it's covering even `compareArrays()`. So in this case, we could really leave `compareArrays()` as private and write just one test. On the other hand, there is no harm in leaving this method public. It helps to have tests to verify that the array comparison works as expected, because it's used as callback for `usort()` and it could be difficult to debug it when we are getting unexpected/unwanted results.

Then the complete implementation looks like the following code snippet:

```
<?php
/**
 * Class SumFinderClass
 */
class SumFinderClass
{
    private $inputArray;

    /**
     * @param $inputArray
     */
    public function __construct($inputArray = null)
    {
        $this->inputArray = $inputArray;
    }

    /**
     * Returns largest sum of contiguous integers
     * @return array
     */
    public function findSum()
    {
        $arrayGroups = array();

        foreach ($this->inputArray as $element) {
            //initial settings
            if (!isset($previousElement)) {
                $previousElement = $element;
                $arrayGroupNumber = 0;
            }

            if (($previousElement + 1) != $element)
```

Tests and What They're All About

```

        $arrayGroupNumber += 1;

        $arrayGroups[$arrayGroupNumber][] = $element;
        $previousElement = $element;
    }

    usort($arrayGroups,array($this,'compareArrays'));
    $highestGroup = array_pop($arrayGroups);

    return $this->extractResult($highestGroup);
}

/**
 * Custom array comparison method to sort by sum of contiguous
integers
 * @param array $a
 * @param array $b
 * @return int
 */
public function compareArrays(array $a, array $b)
{
    $sumA = array_sum($a);
    $sumB = array_sum($b);

    if($sumA == $sumB ) return 0;
    elseif($sumA > $sumB) return 1;
    else return -1;
}

/**
 * @return array|bool
 */
private function extractResult(array $highestGroup)
{
    if(!$highestGroup || !is_array($highestGroup))

        return false;

    $group = implode(' ', $highestGroup);
    $groupSum = array_sum($highestGroup);

    return(array('group'=> $group,'sum'=>$groupSum));
}

```

When you run our test for this class, the following result must be the output:

OK (2 tests, 4 assertions)

As you can see, there is one extra private method `extractResult()`, but because it's an internal implementation inside the class, we are not writing extra tests for this method as it's covered by `testFindSum()`.

The MVC application architecture and tests

After looking at how to test a single piece of functionality, you may ask, what about the whole web application? As mentioned earlier, there are the following levels of testing:

- Unit testing
- Integration testing
- Functional testing

It is important to consider this when you start writing tests. There may be other types of testing, but let's focus on these three for now. When talking about web applications, you will need all of them but different ones in different scenarios.

As you probably know, the design pattern of MVC is used by many web applications and frameworks.

The model is the part where all the business (main) logic is stored. You should definitely have covered with unit tests the main business logic, possibly without database interaction or API calls. This is sometimes a problem with PHP applications; businesses logic equals database access, which is not always right. A bit of abstraction is not a bad idea, things such as **Object Relational Mapping (ORM)** and systems such as Doctrine ORM or Propel really help when it comes to testing.

As a starting point, you should write plain PHPUnit tests with no database interaction, just to cover the main business logic (for example, VAT calculation).

The next thing we are going to explore in detail is integration testing. When you are accessing a database or calling API, it is good to know how your code will interact with another system, and if your implementation is matching, what is happening on the other side.

Compared with unit tests, integration tests could be slow – you probably can't run them as often as unit tests. If you are working with a database, then it depends how you set your test. If you are working on a known dataset and database structure, the database structure can change without your knowledge (then it might be good that the test fails as a changed database structure and not updated code could be a very serious problem). If you work directly with a third-party API, you never know when it might go down or it changes. But again, it could be good to know immediately that something is not quite right.

Controllers shouldn't contain any business logic. If they do, it's usually very ugly spaghetti code, where the code is duplicated, inconsistent, and maybe even full of bugs. A controller should just handle (dispatch) requests and send responses. In theory, you can write unit tests for them, but usually, you use unit testing support provided by MVC frameworks. As you could be launching a whole application, to be able to test controller functionality, you need functional testing. Hundreds or possibly thousands of lines of code are executed to test even simple requests/responses.

And last but not least is a view. A view should just process and display output; nothing else. To keep a strict MVC structure and divide work between the frontend and backend developers, it is good to use a templating system such as Twig or Smarty for the view. With plain PHP, it is very tempting to do more than is necessary. Even view can be tested but usually just by functional tests through controller or by tools such as Selenium when you run black box testing directly in the browser.

The conclusion, when talking about the MVC design pattern, should be that unit testing should focus on a model and a more strict MVC pattern should be applied. The code is going to be of better quality, more easily tested, and possibly will contain less bugs than nasty spaghetti code, where a controller is the master of everything. But you can test controllers, and modern frameworks usually have helpers to allow you to test controllers.

Testing controllers

To see how you can test controllers, let's have a look at what some of the best known PHP MVC frameworks provide. This is just a short overview; you might need the official documentation to see exactly what each framework provides.



The following code snippets show how to test controllers in some popular MVC frameworks, but they won't work without installed and configured frameworks.

The test for Zend Framework 1 can appear as shown in the following code snippet:

```
<?php
class Zf1Test extends Zend_Test_PHPUnit_ControllerTestCase
{
    public function setUp()
    {
        $this->bootstrap = array($this, 'appBootstrap');
        parent::setUp();
    }

    public function testIndexActionShouldContainLoginForm()
    {
        $this->dispatch('/');
        $this->assertAction('index');
        $this->assertResponseCode(200);
        $this->assertQueryContentContains('h1', 'Hello World!');
    }
}
```

The test for Zend Framework 2 can be shown as the following code snippet:

```
<?php

namespace ApplicationTest\Controller;

use Zend\Test\PHPUnit\Controller\AbstractHttpControllerTestCase;

class Zf2Test extends AbstractHttpControllerTestCase
{
    public function setUp()
    {
        $this->setApplicationConfig(include
            '/path/to/application/config/test/
            application.config.php'
        );
        parent::setUp();
    }

    public function testIndexActionCanBeAccessed()
    {
        $this->dispatch('/');
        $this->assertResponseStatusCode(200);

        $this->assertModuleName('application');
        $this->assertControllerClass('IndexController');
        $this->assertActionName('index');
        $this->assertQueryContentContains('h1', 'Hello World!');
    }
}
```

The test for Symphony 2 is shown in the following code snippet:

```
<?php
namespace Application\Tests\Controller;

use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class IndexControllerTest extends WebTestCase
{
    public function testIndex()
    {
        $client = static::createClient();

        $crawler = $client->request('GET', '/');

        $this->assertTrue($client->getResponse()->isSuccessful());

        $this->assertGreaterThan(0,
            $crawler->filter('html:contains
                ("Hello World!")')->count());
    }
}
```

Testing controllers can be very useful, for example, when writing an API. An API can be tested quite easily and should be tested. But to test controllers in this way can be very expensive, as for each test the whole framework and application have to be started again and again, which takes time and resources. You might be tempted to take this approach because when the full application is started, you have all available resources, including a connection to the database.

Summary

This chapter was a quick introduction to unit testing, but the best way to learn how to swim is jumping into the water. That's why we have looked at more than just a couple of PHPUnit tests.

The *Understanding unit testing* section was important to understand why tests are written, how to write good tests, and what the limitations are. The basic PHPUnit test structure was followed by an example of how to test functions and how to test classes and their methods.

But before you start to test anything, you have to think about your application structure and how the MVC design pattern helps with unit testing. As most PHP applications are web applications, we took a short visit to the world of MVC frameworks to see how testing controllers is supported and what you can expect.

In the next chapter, we are going to look at one of the biggest problems when writing tests: dependencies.

4

Database Testing

So far, we have been trying to write unit tests and isolate the tested code. The unit tests are the best approach, and you should always start with unit testing, but when you are testing code, sooner or later you will come across code that needs to read or write to a database. Database testing is tricky, and there are several options to test it. You have to decide on which level you need to test code. The problem comes with database complexity. Database and data in the database are as important as working code, and you need to be sure that everything works together.

The following different approaches can be tried to test a database:

- Use mocking to replace code that connects to the database
- Use a specific database driver only for tests
- Set up a database for integration tests and run tests against this database
- Use a complete development environment and run functional tests against this database

Mocking could be really handy when you need to verify an internal functionality without storing and reading data from a database. A specific database driver could be the solution for some projects that use a custom database abstraction layer, for example, applications that use the MySQLi extension, but for tests, you want to use the SQLite database. The reason to use SQLite is to have a lightweight, fast solution for test execution. Therefore, you might need a specific implementation that allows you to switch to a different database engine.

PHPUnit has a DBUnit extension that allows you to set a database in a known state (clear the database and import data before running tests). This is really handy for database integration tests; you know what is in the database and you can verify changes done by the tests.

The final option is to run tests against the full development database. These tests would then be called functional tests because the database is not in a known state. Testing is focused on code, very often forgetting how a database could be a complex and important part of an application.

Which database to use

Which database to use is a simple yet very important question. As mentioned earlier, a database can be a very complex thing, and the environment for tests must mirror the production environment settings as closely as possible. You should never ever run tests against the production database. Accidentally dropping all tables might not be what you want.

For integration testing, you need a clean database structure, including mandatory data. Surprisingly, this crucial information is often missing. When talking about database structures, a database should be an independent, rock-solid block. Don't look at the compatibility with other database systems, don't look at the limitations that your database persistent layer has, but design a database well. Use database features such as referential integrity definitions (foreign keys) and unique indexes, normalize a database, and when it's convenient, use views and database triggers. Very often, you will see code doing everything, and the database is just a hole to dump data in. A good database is not just a hole. Often, you can do things more easily at the database level. For example, instead of having a super complex SQL query, you can use a stored procedure.

In this chapter, as in the previous chapters, we will use a simplified example that will demonstrate a situation that you might face, and see what we can do.

Going back to the question of which database to use, the answer depends on what your production system is and the way in which you access the database. If your production database is MySQL and you use the `PDO_MySQL` extension, then PDO allows you to use a lightweight database such as SQLite to run tests against a different database engine. The advantage is that in this way, you can verify that the system will work with different database engines. The disadvantage is that you might miss the MySQL behavior that is going to be based on database-specific features such as stored procedures, triggers, and views. Another aspect that many developers just ignore is database permissions, thinking that the `GRANT ALL` statement or root user fixes everything. It does, until you realize that the code you wrote is dependent on the `DROP` permission (enable databases, tables, and views to be dropped), which a user doesn't have on the production system.

The conclusion is that when you write tests, try to make your development environment as close as possible to the production system. Virtualization is a great way to achieve it, and you have to keep the database structure up to date. It depends on which system you use in order to track and deploy database changes, but if the database structure is not kept up to date, then it could be very frustrating to chase and fix bugs that don't exist. When talking about data, the PHPUnit extension, DBUnit, can really help with setting up and cleaning data for tests, but even these datasets need to be kept up to date.

Yes, this can be time-consuming, and in addition to the development environment, you need to keep the test environment, staging environments, and of course production environments up to date. Then, you need to decide which parts are critical and necessary to invest in writing and keeping integration tests running.

Tests for a database

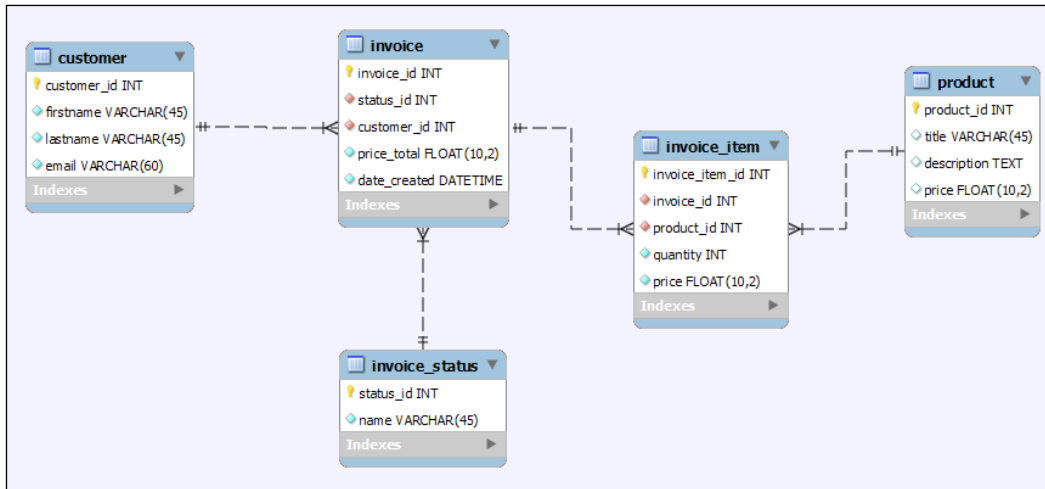
A better way to test a database is to look at one real-life example where we need to sort out a task: a customer buys N products and we need to raise and store an invoice in a database. This is quite a common situation if you create an e-shop or anything similar.

As money is involved, we need to be very careful. We need to test the calculations to ensure that not only are all numbers correct, but also operations that involve storing and obtaining data from a database perform as expected and data is not modified. This is going to be a core functionality that should be covered by tests as much as possible. Every mistake there could be a very expensive mistake.

With similar tasks, it might be a good idea to start with database design if the database is not already available. This usually answers the question about what data is available and how to store it. As mentioned earlier, it is a good idea to set very tight rules about how data is stored. There is nothing more annoying than spending half a day debugging your code and trying to figure out where the problem is, and then realizing that there is nothing wrong with the code but the problem is caused by inconsistent data in the database.

Database Testing

In our case, we used the MySQL Workbench tool (<http://www.mysql.com/products/workbench>) to create the following EER diagram of the database structure:



The advantage of this approach is that you can visualize how your data will be stored, and then export SQL scripts to create tables with all the required indexes and foreign keys.

As you can see, the main table for us is the `invoice` table. Each item in the `invoice` table is linked to the `customer` table, has a status (issued, paid, and canceled), and can have multiple invoice items. The `invoice_item` table is linked to the `product` table. Then, we can export (synchronize) the diagram to SQL. It depends on which database you use, but PHP is most commonly used together with MySQL or MariaDB. The following example is for MySQL and MariaDB, but this can be easily converted to any similar RDBMS database:

```

CREATE TABLE 'customer' (
  'customer_id' int(11) NOT NULL AUTO_INCREMENT,
  'firstname' varchar(45) NOT NULL,
  'lastname' varchar(45) NOT NULL,
  'email' varchar(60) NOT NULL,
  PRIMARY KEY ('customer_id'),
  UNIQUE KEY 'email_UNIQUE' ('email')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
  
```

```

CREATE TABLE 'invoice' (
    'invoice_id' int(11) NOT NULL AUTO_INCREMENT,
    'status_id' int(11) NOT NULL,
    'customer_id' int(11) NOT NULL,
    'total_gross' float(10,2) NOT NULL,
    'total_net' float(10,2) NOT NULL,
    'total_vat' float(10,2) NOT NULL,
    'date_created' datetime NOT NULL,
    PRIMARY KEY ('invoice_id'),
    KEY 'fk_invoice_customer_idx' ('customer_id'),
    KEY 'fk_invoice_invoice_status1_idx' ('status_id'),
    CONSTRAINT 'fk_invoice_customer' FOREIGN KEY ('customer_id')
    REFERENCES 'customer' ('customer_id') ON DELETE NO ACTION ON
    UPDATE NO ACTION, CONSTRAINT 'fk_invoice_invoice_status1'
    FOREIGN KEY ('status_id') REFERENCES 'invoice_status'
    ('status_id') ON DELETE NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE 'invoice_status' (.....);

CREATE TABLE 'product' (.....);

CREATE TABLE 'invoice_item' (.....);

```

This SQL code creates the required database structure. As you can see, foreign keys are in place because they are part of any decent database structure.

When we have the database structure in place, we can start thinking of how to fill this structure with the required data. While working with databases, it is handy to turn data into objects and work with objects in PHP. It is then clear as to when and how to store/load data from a database, but the data structure can still be used in a PHP code without too much hassle. For this example, we just use PDO, but later, we will see how to use Doctrine ORM, which works as a bridge between PHP code and a database.

To make the code cleaner, every table is represented as a class, as shown in the following code snippet. We are going to use these classes while fetching data by the fetch mode `PDO::FETCH_CLASS` and when the returned data is going to be bound to these classes.

```

class Customer {

    public $customer_id;
    public $firstname;
    public $lastname;
    public $email;

}

```

Database Testing

```

class Product {

    public $product_id;
    public $title;
    public $description;
    public $price;

}

class Invoice {

    const STATUS_ISSUED = 1;
    const STATUS_PAID = 2;
    const STATUS_CANCELLED = 3;

    public $invoice_id;
    public $status_id;
    public $customer_id;
    public $price_total = 0;

    public $date_created;
    /**
     * @var InvoiceItem[]
     */
    public $invoiceItems = array();
}

class InvoiceItem {

    public $invoice_item_id;
    public $invoice_id;
    public $product_id;
    public $quantity;
    public $price;

}

```

Now, the task is to create an invoice, store the invoice in the database, and then verify that the data is stored correctly. For this, we will create a class, *InvoiceManager*, which will provide the required functionality.

As our code will mainly communicate with the database, in this case, we jump directly into writing the code. This is possible, but please do remember that you have to write tests, if not before writing the code, at least on the same day when you write the code. It is very easy to slip back to the habit that code somehow works: now I have something more urgent, I will write the test later – and later almost equals never.

In our example, we created the `InvoiceManager` class that will store and load invoices from the database. A simple `InvoiceManager` class that uses the PDO driver could look like the following code snippet:

```
class InvoiceManager{
    public function __construct(PDO $db){}
    public function loadInvoice($invoiceId){}
    public function raiseInvoice(Invoice $invoice, Customer
        $customer, array $productsArray){}
    protected function storeInvoice(Invoice $invoice){}
}
```

It is now important to think about what the class will do. We pass the database connection to the constructor. The `raiseInvoice` method creates the `invoice` and `invoiceItem` objects based on the passed arguments: `invoice`, `customer`, and `productsArray`. You can imagine `productsArray` as an array of products and quantities, as shown in the following line of code:

```
array('product' => $product1, 'quantity' => 1),
array('product' => $product2, 'quantity' => 2)
```

Here, `product` is an instance of the `Product` class, and `quantity` indicates how many of these products were ordered. Each of these products then needs to be turned into `InvoiceItem`. When the total values for the invoices are set, we can store the invoices in the database.

The complete, simple implementation of the `InvoiceManager` class could look like the following code snippet:

```
<?php

class InvoiceManager
{
    private $db;

    public function __construct(PDO $db)
    {
        $this->db = $db;
    }
}
```

The PDO object is passed to the constructor. For testing, a dependency is important. If you have a dependency inside your class, then it might be very difficult to replace it with something else. The `loadInvoice()` method uses the passed PDO object to the constructor and performs database calls:

```
(c) Copyright 2016 OOB    public function loadInvoice($invoiceId)
                        {
```

Database Testing

```

        $sql = 'SELECT * FROM invoice where invoice_id =
                :invoice_id';
        $stm = $this->db->prepare($sql);
        $stm->execute(array(':invoice_id' => $invoiceId));
        $stm->setFetchMode(PDO::FETCH_CLASS, 'Invoice');
        $invoice = $stm->fetch();

        $sql = "SELECT * FROM invoice_item where invoice_id =
                :invoice_id";
        $stm = $this->db->prepare($sql);
        $stm->execute(array(':invoice_id' => $invoiceId));
        $stm->setFetchMode(PDO::FETCH_CLASS, 'InvoiceItem');

        while ($invoiceItem = $stm->fetch())
        {
            $invoice->addInvoiceItem($invoiceItem);
        }

        return $invoice;
    }

```

The `loadInvoice()` method is a simple method to load an invoice from the database and binding results to objects. To keep examples short, there is no added error handling mechanism, for example, when an invoice or invoice item is not found. The `raiseInvoice()` method creates an invoice from passed parameters:

```

    public function raiseInvoice(Invoice $invoice, Customer
        $customer, array $productsArray)
    {
        $invoice->customer_id = $customer->customer_id;
        $invoice->status_id = Invoice::STATUS_ISSUED;
        $invoice->date_created = new DateTime();

        foreach ($productsArray as $productItem)
        {
            $product = $productItem['product'];
            $quantity = $productItem['quantity'];

            $invoiceItem = new InvoiceItem();
            $invoiceItem->product_id = $product->product_id;
            $invoiceItem->quantity = $quantity;
            $invoiceItem->price = $product->price;

            $invoice->addInvoiceItem($invoiceItem);
        }
    }

```

```

        $invoice->setTotals();
        return $this->storeInvoice($invoice);
    }

```

The `raiseInvoice()` method translates the ordered products into an invoice, calculates the total value of the invoice, and then calls the `storeInvoice()` method to store data into the database, as shown in the following code snippet:

```

protected function storeInvoice(Invoice $invoice)
{
    $sql = "INSERT INTO invoice (status_id, customer_id,
        price_total, date_created) VALUES (:status_id,
        :customer_id, :price_total, :date_created)";
    $stm = $this->db->prepare($sql);

    $stm->execute(array(':status_id' => $invoice->status_id,
        ':customer_id' => $invoice->customer_id, ':price_total'
        => $invoice->price_total, ':date_created' =>
        $invoice->date_created->format("Y-m-d H:i:s")));

    $invoiceId = $this->db->lastInsertId();
    if (!$invoiceId) throw new Exception('Invoice not saved');
    $invoice->invoice_id = $invoiceId;

    foreach ($invoice->invoiceItems as $invoiceItem)
    {
        $invoiceItem->invoice_id = $invoiceId;

        $sql = "INSERT INTO invoice_item (invoice_id,
            product_id, quantity, price) VALUES (:invoice_id,
            :product_id, :quantity, :price)";
        $stm = $this->db->prepare($sql);

        $stm->execute(array(':invoice_id' =>
            $invoiceItem->invoice_id,
            ':product_id' => $invoiceItem->product_id,
            ':quantity' => $invoiceItem->quantity, ':price'
            => $invoiceItem->price));
    }

    return $invoice;
}

```

Database Testing

To have a complete example that we are going to use for testing and see different testing methods, let's take a look at the `Invoice` class to see all its methods properly, as shown in the following code snippet:

```
public function addInvoiceItem(InvoiceItem $item)
{
    $this->invoiceItems[$item->product_id] = $item;
}

public function removeInvoiceItem(InvoiceItem $item)
{
    unset($this->invoiceItems[$item->invoice_id]);
}

public function setTotals()
{
    $this->price_total = 0;

    foreach ($this->invoiceItems as $invoiceItem)
    {
        $this->price_total += $invoiceItem->price *
            $invoiceItem->quantity;
    }
}
```

To test the `InvoiceManager` class, you have two options: unit tests by using mocking techniques, which we learned in *Chapter 8, Using Test Doubles*, or integration tests where you call a database and you also verify interactions with the database. This data is stored/retrieved from the database.

Now, let's take a look at how to write tests using mocks. As you can see, a database interacts with the constructor through the passed PDO object, so for testing purposes, you need to replace the PDO object with mock. With mocking PDO, there is one known problem. When you run it, you will get the following error:

PDOException: You cannot serialize or unserialize PDO instances.

It doesn't help even to disable the constructor, but there is a simple solution; for the test, you need to extend the `PDO` class and replace its constructor in the child class, as shown in the following lines of code:

```
<?php
```

```
class PDOMock extends PDO
{
```

```

        public function __construct() {}
    }

```

This does the trick. Now, we can use the mock object to replace the PDO class object as shown in the following code snippet:

```

<?php

class InvoiceManagerTest extends PHPUnit_Framework_TestCase
{
    private $pdoMock;
    private $stmMock;

    public function setUp()
    {
        $this->stmMock = $this->getMock('PDOStatement',
            array('execute', 'fetch'));
        $this->stmMock->expects($this->any())->method('execute')
            ->will($this->returnValue(true));

        $this->pdoMock = $this->getMock('PDOMock',
            array('prepare', 'lastInsertId'));
        $this->pdoMock->expects($this->any())->method('prepare')
            ->will($this->returnValue($this->stmMock));
    }
}

```

In the `setUp()` method, we created the mock object for the PDO class, or, to be precise, for our extended class `PDOMock`. As the PDO `prepare()` method returns the `PDOStatement` object, we also need to replace this object to imitate database calls. Instead of calling the database using the PDO object, we use our mock object passed to the `InvoiceManager` constructor:

```

    public function testRaiseInvoice()
    {
        $this->pdoMock->expects($this->once())
            ->method('lastInsertId')->will($this->returnValue(1));

        $invoiceManager = new InvoiceManager($this->pdoMock);

        $product1 = new Product();
        $product1->price = 10;
        $product1->product_id = 1;

        $customer = new Customer();
        $customer->customer_id = 1;
    }
}

```

Database Testing

```

        $invoice = new Invoice();
        $productsArray = array(array('product' => $product1,
            'quantity' => 2));
        $invoiceManager->raiseInvoice($invoice, $customer,
            $productsArray);

        $this->assertEquals(20, $invoice->price_total);
    }

```

This test calls the `raiseInvoice()` method, which stores data into the database. So for us, it is enough to just use `stmMock`, which is the mock of `PDOStatement`, and replace the `execute()` method with the one returning `true`, and `lastInsertId()` method with the one returning `1`. This way we can exercise the code and test for the `loadInvoice()` method, which could look like the following code snippet:

```

public function testLoadInvoice()
{
    $invoice = new Invoice();
    $invoice->invoice_id = 1;
    $invoice->price_total = 100;

    $invoiceItem = new InvoiceItem();
    $invoiceItem->invoice_item_id = 1;
    $invoiceItem->invoice_id = 1;
    $invoiceItem->price = 100;
    $invoiceItem->product_id = 1;

    $this->stmMock->expects($this->at(1))->method('fetch')
        ->will($this->returnValue($invoice));

    $this->stmMock->expects($this->at(3))->method('fetch')
        ->will($this->returnValue($invoiceItem));

    $invoiceManager = new InvoiceManager($this->pdoMock);
    $invoice = $invoiceManager->loadInvoice(1);

    $this->assertEquals(100, $invoice->price_total);
    $this->assertEquals(100, $invoice->invoiceItems[1]
        ->price);
}
}

```

The test `testLoadInvoice()` might look a bit tricky. When you look at the `InvoiceManager` class code, it is calling the method `fetch()` twice while loading data from the database. We need to replace these methods, but both should return different results, first `invoice` and second `invoiceItem`. To be sure that different results are returned, we used `expects($this->at())` and the index number. The reason why the index is one and three, and not zero and one, as you might expect, is that PHPUnit is counting calls on mocked methods for `execute` and `fetch` on the mocked `PDOStatement` object. First, `execute` is called and then `fetch`, followed by the second `execute` and `fetch` methods.

To ensure that different results are returned, another option would be to use `returnValueMap` or `returnCallback`.

When we run the unit test, everything looks good, and we get the following result:

PHPUnit 3.7.30 by Sebastian Bergmann.

Time: 87 ms, Memory: 4.75Mb

OK (2 tests, 7 assertions)

At this stage, testing really helps. When you write code that has more than 10 lines, then there is a really good chance of you making at least one mistake. When you write code like this, which has 160 lines, then there are going to be a few mistakes. At this stage, tests allow you to immediately verify and fix the written code, and the initial investment in writing tests pays off. If you allow buggy code to be used, then you are going to spend ages testing applications in the browser, going back, fixing one problem, pushing a new release to the testing environment, and then having another reported bug. In the worst case, the customer can be used as the tester, and the initial impression is not going to be good.

When we look back at our code, we have unit tests, but the problem is the database. The `storeInvoice()` and `loadInvoice()` methods are also crucial parts of the code. If the invoice is not going to be stored in the database as we expect, numbers will be changed. Then, it's as serious a problem as doing the wrong calculations. This is finally the moment when we move to integration testing.

The point of integration testing is to verify how our code will interact with the database. We need to be sure that all the insert, select, and update statements are executed. We are not going to receive any database errors, and the data will be stored/loaded correctly.

DBUnit

DBUnit is a PHPUnit extension that makes our integration testing easier. DBUnit allows you to clear the database and import a prepared dataset to set the database to a known state. This is important for database testing. If you set up a test database and allow data to be stored there without cleaning it, sooner or later you will have negative fails, and your test will be less reliable.

To make it clear, DBUnit helps with database testing, but it's not a magical tool that will solve all problems. You need to create and maintain your database's structure. In addition, you need to maintain datasets used for tests. I don't remember the last time I worked on a project that didn't use a database abstraction layer. When you use a database abstraction layer, it is better to rely on your implementation rather than on DBUnit's assert features, but DBUnit is still a good starting point and a big help.

Installing DBUnit

You should follow the installation method used for your PHPUnit's installation.

For PEAR installation, run the following command line:

```
>pear install phpunit/DbUnit
```

For the Composer installation, modify `composer.json` as shown in the following code snippet:

```
{
    "require": {
        "phpunit/phpunit": "3.7.*",
        "phpunit/dbunit": "1.2.*"
    }
}
```

Database test cases

When we wish to use DBUnit, we have to make one change. When writing the unit test, our test case extends `PHPUnit_Framework_TestCase`. When using DBUnit, our test case has to extend `PHPUnit_Extensions_Database_TestCase`. This class has the following two abstract methods, which we have to implement:

- `getConnection()`: This method must return `PHPUnit_Extensions_Database_DB_IDatabaseConnection`
- `getDataSet()`: This method must return `PHPUnit_Extensions_Database_DataSet_IDataSet`

The first method returns a database connection but not directly a PDO or MySQLi connection; it is a connection wrapped in the `PHPUnit` class. However, don't worry; usually it's easy; just specify the database server, username, password, and database name, as shown in the following line of code:

```
return $this->createDefaultDBConnection
    (new PDO('mysql:host=localhost;dbname=book','root','password'));
```

Otherwise, the database's test case is the same test case as the normal test case with a few extra assertions.

Datasets

Datasets are probably the most useful DBUnit feature. You can define a set of data to be included in the database. Data from tables that are to be populated by a dataset are truncated. This is called through the `setUp()` method before every test is executed. This is great for restoring a database to a known state, but as you can guess, it's not going to be the fastest thing in the world.

DBUnit supports datasets in several different formats, which are as follows:

- **A flat XML dataset:** This is an XML format where each row is a record in the table, and the columns are attributes
- **An XML dataset:** This is a structured XML format and is more complex than a flat XML dataset
- **A MySQL XML dataset:** This is created with the `mysqldump` utility using the `--xml` switch
- **A CSV dataset:** This is a dataset in the CSV format
- **A YAML dataset:** This is a dataset in the YAML format

All datasets are very similar. They are just in a different format. NULL values may be problematic as they are not handled very well by the flat XML format and are better handled, for example, by the YAML format.

The content of these basic datasets can be modified by other datasets, which use them as inputs and then perform some operations before returning a modified dataset, as follows:

- **Replacement dataset:** This applies rules to modify the dataset's content and helps to overcome the XML problems with NULL values
- **Composite dataset:** This merges multiple datasets
- **Dataset filter:** This filters the content of a dataset

These are static datasets created with prepared data, but DBUnit also offers dynamic datasets. These datasets are created on the fly, for example, as the result of a query, and they are handy when you want to compare data.

Using DBUnit

A good way to understand how DBUnit works is by going back to our invoice example and seeing what we can do. So far, we have created a unit test that verified the calculations. What remains to be done is testing the `loadInvoice()` and `storeInvoice()` methods to verify that they are stored and loaded from the database correctly.

If you look back at the database's structure, you will see that the invoice status is required and sits in the `invoice_status` table, so this is definitely a table that needs to be populated. To store an invoice, we need the customer ID, and for invoice items we need products, and we need more two tables, `customers` and `products`, which need to be populated. It would then be good to insert some dummy invoice data there to ensure that the invoice is inserted correctly.

Probably one of the easiest ways to use datasets is to use a flat XML dataset. So let's have a look at what this dataset could look like:

```
<?xml version="1.0"?>
<dataset>
  <customer customer_id="1" firstname="John" lastname="Smith"
    email="john.smith@localhost"/>
  <customer customer_id="2" firstname="Jenny" lastname="Smith"
    email="jenny.smith@localhost"/>
  <product product_id="1" title="test product 1"
    description="some description" price="10.00"/>
  <product product_id="2" title="test product 2"
    description="some description" price="20.00"/>
  <invoice_status status_id="1" name="issued"/>
  <invoice_status status_id="2" name="paid"/>
  <invoice_status status_id="3" name="canceled"/>
  <invoice invoice_id="1" status_id="1" customer_id="1"
    price_total="100.00" date_created="2013-01-20 08:00:00"/>
  <invoice_item invoice_item_id="1" invoice_id="1"
    product_id="1" quantity="1" price="100.00"/>
</dataset>
```

As you can see, each dataset element is a table name, and the attributes match the column names. In this way, we define all data that needs to be inserted into the database. Each table mentioned in the dataset will be truncated.

To see how a dataset works and what you can do with it, let's use a simple test as shown in the following code snippet:

```
<?php
class DatasetTest extends PHPUnit_Extensions_Database_TestCase
{
    protected $connection = null;

    public function getDataSet()
    {
        return $this->createFlatXmlDataSet('dataset.xml');
    }
}
```

The `getDataSet()` method loads an XML dataset from our flat XML file that contains data to be injected into the database:

```
protected function setUp()
{
    $conn=$this->getConnection();
    $conn->getConnection()->query("set foreign_key_checks=0");
    parent::setUp();
    $conn->getConnection()->query("set foreign_key_checks=1");
}
```

The `setUp()` method disables foreign key checks and then injects the data from the dataset into the database. The reason why the foreign keys are disabled is that MySQL would refuse to store the data. Another solution to avoid referential integrity problems might be to order data in the XML dataset that will be inserted in order to avoid referential integrity problems:

```
protected function getConnection()
{
    if ($this->connection === null)
    {
        $connectionString =
            $GLOBALS['DB_DRIVER'].':host='.$GLOBALS['DB_HOST'].
            ';dbname='.$GLOBALS['DB_DATABASE'];
        $this->connection =
            $this->createDefaultDBConnection(
                new PDO($connectionString,
                    $GLOBALS['DB_USER'], $GLOBALS['DB_PASSWORD']));
    }

    return $this->connection;
}
```

Database Testing

The `getConnection()` method creates the PDO database connection for tests. It uses the `$GLOBALS` variable and stores GLOBALS configuration, such as `$GLOBALS['DB_HOST']`, which is taken from the configuration file:

```
public function testConsumer()
{
    $stm = $this->getConnection()->getConnection()
        ->prepare("select * from customer where customer_id =
            :customer_id");

    $stm->execute(array('customer_id' => 2));
    $result = $stm->fetch();

    $this->assertEquals("jenny.smith@localhost",
        $result['email']);

    $dbTable = $this->getConnection()->createQueryTable(
        'customer', 'SELECT * FROM customer');

    $datasetTable = $this->getDataSet()
        ->getTable("customer");

    $this->assertTablesEqual($dbTable, $datasetTable);
}
```

The `testConsumer()` method uses the `createQueryTable()` method to load the data from the database and `assertTablesEqual` to verify that the data is the same as the data in the dataset:

```
public function testInvoice()
{
    $dataSet = new
        PHPUnit_Extensions_Database_DataSet_QueryDataSet
        ($this->getConnection());
    $dataSet->addTable('customer');
    $dataSet->addTable('product');
    $dataSet->addTable('invoice');
    $dataSet->addTable('invoice_item');
    $dataSet->addTable('invoice_status');
    $expectedDataSet = $this->getDataSet();

    $this->assertDataSetsEqual($expectedDataSet, $dataSet);
}
```

The `testInvoice()` method is similar, except that it compares not just one table but all tables with the dataset.

The `assertDataSetsEqual()` and `assertTablesEqual()` methods are handy when you need to verify data, and you can use sets of predefined datasets.

When you look at this code, you will see a few differences compared to the usual unit test, and these are as follows:

- It extends `PHPUnit_Extensions_Database_TestCase`
- It implements `getDataSet()` and `getConnection()`
- The `setUp()` method contains DB queries
- It does not use mocks

When the test is executed, the `setUp()` method is called. We need to use the SQL query to disable foreign key checks on a MySQL level. DBUnit is just trying to truncate tables, and when you have foreign keys in place, MySQL would refuse to truncate them without disabling foreign key check. For a connection, we use the `getConnection()` method that creates a database connection, and the `getDataSet()` method is called.

The `getDataSet()` method returns a flat XML dataset from our XML configuration file. DBUnit processes this file, truncates tables, and inserts data into the database, and then the test is executed.

For the connection details, we used the `$GLOBALS` variable. You might wonder where the data came from. Instead of hardcoding it in tests, you can store it in a configuration file, usually `phpunit.xml`:

```
<?xml version="1.0" encoding="UTF-8" ?>
<phpunit bootstrap="bootstrap.php">
  <php>
    <var name="DB_DRIVER" value="mysql" />
    <var name="DB_USER" value="root" />
    <var name="DB_PASSWORD" value="password" />
    <var name="DB_HOST" value="localhost" />
    <var name="DB_DATABASE" value="book" />
  </php>
</phpunit>
```

Database Testing

We can use the `DatasetTest` class because it has everything that we need to test the database, extend it, and create integration tests for the `InvoiceManager` class, as shown in the following code snippet:

```
<?php

class InvoiceManagerDBTest extends DatasetTest
{
    public function testRaiseInvoice()
    {
        $invoiceManager = new InvoiceManager(
            $this->getConnection()->getConnection());

        $product1 = new Product();
        $product1->price = 10;
        $product1->product_id = 1;

        $customer = new Customer();
        $customer->customer_id = 1;

        $invoice = new Invoice();
        $productsArray = array(array('product' => $product1,
            'quantity' => 2));
        $invoiceManager->raiseInvoice($invoice, $customer,
            $productsArray);

        $invoiceFromDB = $invoiceManager
            ->loadInvoice($invoice->invoice_id);

        $this->assertEquals($invoice->price_total,
            $invoiceFromDB->price_total);
        $this->assertEquals(count($invoice->invoiceItems),
            count($invoiceFromDB->invoiceItems));
    }
}
```

We are not using mocks because in this case we want to test the entire functionality. The `raiseInvoice()` method inserts a new record into the database and then uses `loadInvoice()` to load the newly created invoice. When you execute the test case, you should see an output similar to the following command line:

PHPUnit 3.7.30 by Sebastian Bergmann.

Time: 444 ms, Memory: 5.00Mb

(c) Copyright 2016 QOS
OK (3 tests, 5 assertions)

Doctrine 2 ORM and database testing

When you are working on a modern application, you almost use a database abstraction layer. As an example of how to use DBUnit with a third-party library, let's have a look at how to use Doctrine 2 ORM, which is a heavyweight but very powerful object-relational mapper using our own **Database Abstraction Layer (DAL)**. Doctrine is quite complex, and the aim is not to explain how it works, but to show you how to use database integration testing with other database abstraction layers.

To be able to run code, you have to install the required libraries. In this case, it is strongly suggested that you use the Composer installation and a class loader to install Doctrine by setting the required dependency in the `composer.json` file. Along with Doctrine ORM, there is also added `beberlei/DoctrineExtensions` for the `OrmTestCase` class, which makes testing easier:

```
{
  "require": {
    "doctrine/orm": "2.4.*",
    "beberlei/DoctrineExtensions": "dev-master"
  },
  "autoload": {
    "psr-0": { "": "src/" }
  },
  "minimum-stability" : "dev"
}
```

This sets the required dependency in the `composer.json` file, and also loads the `OrmTestCase` class, which makes testing easier.

When you work with Doctrine, instead of accessing database tables, you work with entities that are mapped to the database tables. When required, `EntityManager` persists these entities and stores them in the database. You are not touching the database, and all the database interaction is done through these entities. As an example, you can see the `Invoice` entity's properties and mapping declarations, which are done through annotations. All other entities can be found on https://github.com/machek/PHPUnit_Essentials.

```
<?php

namespace MyEntity;

use Doctrine\ORM\Mapping as ORM;
```

Database Testing

```

/**
 * Invoice
 *
 * @ORM\Table(name="invoice")
 * @ORM\Entity
 */
class Invoice
{
    /**
     * @var integer
     *
     * @ORM\Column(name="invoice_id", type="integer", nullable=false)
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="IDENTITY")
     */
    private $invoiceId;

    /**
     * @var \DateTime
     *
     * @ORM\Column(name="date_created", type="datetime",
    nullable=false)
     */
    private $dateCreated;

    /**
     * @var \MyEntity\Customer
     *
     * @ORM\ManyToOne(targetEntity="MyEntity\Customer")
     * @ORM\JoinColumns({
     *     @ORM\JoinColumn(name="customer_id",
     *         referencedColumnName="customer_id")
     * })
     */
    private $customer;

```

.....

As you can see, this is similar to the Invoice class that we used for PDO, but it also describes how it's mapped.

When we look at the `InvoiceManager` class that we used for storing an invoice to the database by using the `PDO` class with `Doctrine`, it looks slightly different, but not too much, as shown in the following code snippet:

```
<?php

use Doctrine\ORM\EntityManager;

class InvoiceManagerDoctrine
{
    /**
     * @var \Doctrine\ORM\EntityManager
     */
    private $em;

    public function __construct(EntityManager $em)
    {
        $this->em = $em;
    }

    /**
     * @param Customer $customer
     * @param array $productsArray
     * @return mixed
     */
    public function raiseInvoice(MyEntity\Invoice $invoice,
        MyEntity\Customer $customer, array $productsArray,
        MyEntity\InvoiceStatus $invoiceStatus)
    {
        $invoice->setCustomer($customer);
        $invoice->setStatus($invoiceStatus);
        $invoice->setDateCreated(new DateTime());

        foreach ($productsArray as $productItem)
        {
            $product = $productItem['product'];
            $quantity = $productItem['quantity'];

            $invoiceItem = new MyEntity\InvoiceItem();
            $invoiceItem->setProduct($product);
            $invoiceItem->setQuantity($quantity);
            $invoiceItem->setPrice($product->getPrice());
            $invoice->addInvoiceItem($invoiceItem);
        }
    }
}
```

Database Testing

```

        $invoice->setTotals();
        return $this->storeInvoice($invoice);
    }

    /**
     * @param $invoiceId
     * @return \MyEntity\Invoice
     */
    public function loadInvoice($invoiceId)
    {
        return $this->em->find('\MyEntity\Invoice', $invoiceId);
    }

    /**
     * @return bool
     * @throws Exception
     */
    protected function storeInvoice(MyEntity\Invoice $invoice)
    {
        foreach($invoice->getInvoiceItem() as $item)
        {
            $this->em->persist($item);
        }

        $this->em->persist($invoice);
        $this->em->flush();
    }
}

```

As you can see, it's quite similar. The only difference is that instead of assigning the IDs, such as `customer_id`, you set the `Customer` entity, and Doctrine handles how it's stored.

Why do you need to see all of this? If you remember, DBUnit is not taking care of your database's structure, it's just importing data from datasets. This is where a different approach could be really useful, but keeping a database structure up to date can be a real pain.

With Doctrine, it's easier. You use entities. Each entity describes how it is mapped to the database. When you want to make any change in the database's structure, you have to update the entity. This way, you know that you always have an up-to-date description of what a database looks like, and then you can recreate the database.

It sounds like magic, but it works.

Let's have a look at an example of how to test our `InvoiceManager` class using Doctrine ORM, as shown in the following code snippet:

```
<?php

require_once '../vendor/autoload.php';

use DoctrineExtensions\PHPUnit\OrmTestCase,
    Doctrine\ORM\Tools\Setup,
    Doctrine\ORM\EntityManager,
    Doctrine\ORM\Tools\SchemaTool;

class InvoiceManagerDoctrineTest extends OrmTestCase
{
    /**
     * @var Doctrine\ORM\EntityManager
     */
    protected static $em = null;

    public static function setUpBeforeClass()
    {
        $config =
            Setup::createAnnotationMetadataConfiguration(array(
                __DIR__ . '/../src'), true, null, null, false);

        $connectionOptions = array('driver' => 'pdo_sqlite',
            'memory' => true);

        // obtaining the entity manager
        self::$em = EntityManager::create($connectionOptions,
            $config);

        $schemaTool = new SchemaTool(self::$em);

        $cmf = self::$em->getMetadataFactory();
        $classes = $cmf->getAllMetadata();

        $schemaTool->dropDatabase();
        $schemaTool->createSchema($classes);
    }

    protected function tearDown()
```

Database Testing

```

    {
        self::$em->clear();
        parent::tearDown();
    }

    protected function createEntityManager()
    {
        return self::$em;
    }

    protected function getDataSet()
    {
        return $this->createFlatXmlDataSet(__DIR__ .
            "../../dataset.xml");
    }

    public function testLoadInvoice()
    {
        $invoiceManager = new \InvoiceManagerDoctrine(self::$em);
        $invoice = $invoiceManager->loadInvoice(1);

        $this->assertEquals(1, $invoice->getInvoiceId());
        $this->assertEquals(1, $invoice->getCustomer()
            ->getCustomerId());
        $this->assertEquals(100, $invoice->getPriceTotal());

        $invoiceItems = $invoice->getInvoiceItem();

        $this->assertEquals(100, $invoiceItems[0]->getPrice());
        $this->assertEquals(1, $invoiceItems[0]->getProduct()
            ->getProductId());
    }

    public function testRaiseInvoice()
    {
        $invoiceManager = new \InvoiceManagerDoctrine(self::$em);

        $product = self::$em->find('\MyEntity\Product', 1);

        $customer = self::$em->find('\MyEntity\Customer', 1);
        $status = self::$em->find('\MyEntity\InvoiceStatus', 1);

        $invoice = new \MyEntity\Invoice();
        $invoiceManager->raiseInvoice($invoice, $customer,

```

```

        array(array('product' => $product, 'quantity'
            => 2)), $status);

$this->assertEquals(20, $invoice->getPriceTotal());

$invoiceFromDB = $invoiceManager->loadInvoice($invoice
    ->getInvoiceId());

$this->assertEquals($invoice, $invoiceFromDB);
    }
}

```

This test case is slightly different, but not much. When we open a connection to the database, we can share it between tests, because the database content is going to be wiped out anyway. For this, we used the `setUpBeforeClass()` method and the protected static `$em` property to share `EntityManager` between tests.

The following code snippet is the most interesting part:

```

$connectionOptions = array('driver' => 'pdo_sqlite',
    'memory' => true);

// obtaining the entity manager
self::$em = EntityManager::create($connectionOptions,
    $config);

$schemaTool = new SchemaTool(self::$em);

$cmf = self::$em->getMetadataFactory();
$classes = $cmf->getAllMetadata();

$schemaTool->dropDatabase();
$schemaTool->createSchema($classes);

```

As Doctrine can use different database engines without any code change, we used `pdo_lite` in the memory database. The code then loads metadata from all entities and creates all tables on the fly. Even though Doctrine is a heavyweight champion, because it runs just through the PHP and is kept in memory, its performance is not bad. The following is the result we get:

```

PHPUnit 3.7.30 by Sebastian Bergmann.
Time: 120 ms, Memory: 13.25Mb
OK (2 tests, 9 assertions)

```

Summary

Database integration tests are very important. When data is not stored correctly in the database or if we can't load data correctly, then even the best code is not good. When you start thinking about integration testing, it sounds really tricky. How do you isolate it? How do you write reliable tests? Mocking helps to isolate the code as a starting point, but when we need to test database interactions, then DBUnit really helps with setting up the database into a known state.

Database integration tests could be as important as unit tests, but they can be slower and sometimes less reliable. Because of these reasons, it is suggested that you keep them separate from unit tests. There are several options of how to do this, which we have seen in *Chapter 7, Organizing Tests*.

In the next chapter, we will see a similar topic, the testing API, which could be even worse than testing the database. The database is yours, you can trust it, but not the third-party API. You need tests to help you develop and verify code.

5

Continuous Integration

You have code and you have tests, but now you need to take complete advantage of them in order for them to really help you. What you need to do is run these tests, process the results, and then receive a notification if they fail. This is where we are heading, and there are a few really good open source or free solutions available that can help you.

This chapter is named **Continuous Integration**; so, what exactly does this mean? You can find many long definitions, but to put it simply, it is a process where you integrate your code with code from other developers and run tests to verify the code functionality. You are aiming to detect problems as soon as possible and trying to fix problems immediately. It is always easier and cheaper to fix a couple of small problems than create one big problem.

This can be translated to the following workflow:

1. The change is committed to a version control system repository (such as Git or SVN).
2. The **Continuous Integration (CI)** server is either notified of, or detects a change and then runs the defined tests.
3. CI notifies the developer if the tests fail.

With this method you immediately know who created the problem and when.

For the CI to be able to run tests after every commit point, these tests need to be fast. Usually, you can do this with unit tests for integration, and with functional tests it might be better to run them within a defined time interval, for example, once every hour.

You can have multiple sets of tests for each project, and another golden rule should be that no code is released to the production environment until all of the tests have been passed.

It may seem surprising, but these rules and processes shouldn't make your work any slower, and in fact, should allow you to work faster and be more confident about the developed code functionality and changes. Initial investment pays off when you can focus on adding new functionality and are not spending time on tracking bugs and fixing problems. Also, tested and reliable code refers to code that can be released to the production environment more frequently than traditional big releases, which require a lot of manual testing and verification. There is a real impact on business, and it's not just about the discussion as to whether it is worthwhile and a good idea to write some tests and find yourself restricted by some stupid rules anymore.

What will really help and is necessary is a CI server for executing tests and processing the results; this is also called **test automation**. Of course, in theory you can write a script for it and test it manually, but why would you do that when there are some really nice and proven solutions available? Save your time and energy to do something more useful.

In this chapter, we will see what we can do with the most popular CI servers used by the PHP community:

- Travis CI
- Jenkins CI
- Xinc

For us, a CI server will always have the same main task, that is, to execute tests, but to be precise, it includes the following steps:

1. Check the code from the repository.
2. Execute the tests.
3. Process the results.
4. Send a notification when tests fail.

This is the bare minimum that a server must handle. Of course, there is much more to be offered, but these steps must be easy to configure.

Using a Travis CI hosted service

Travis is the easiest to use from the previously mentioned servers. Why is this the case? This is because you don't have to install it. It's a service that provides integration with GitHub for many programming languages, and not just for PHP. Primarily, it's a solution for open source projects, meaning your repository must be a public repository. It also has commercial support for private repositories and commercial projects.

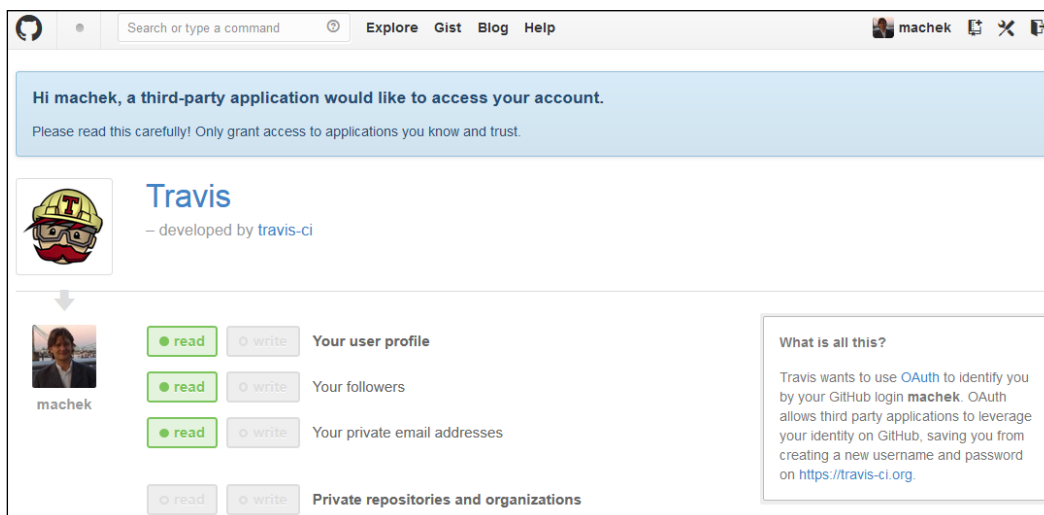
What is really good is that you don't have to worry about server configuration; instead, you just have to specify the required configuration (in the same way you do with Composer), and Travis does everything for you. You are not just limited to unit tests, and you can even specify which database you want to use and run integration tests there.

However, there is also a disadvantage to this solution. If you want to use it for a private repository, you have to pay for the service, and you are also limited with regard to the server configuration. You can specify your PHP version, but it's not recommended to specify a minor version such as 5.3.8; you should instead use a major version, such as 5.3. On the other hand, you can run tests against various PHP versions, such as PHP 5.3, 5.4, or 5.5, so when you want to upgrade your PHP version, you already have the test results and know how your code will behave with the new PHP version.

Travis has become the CI server of choice for many open source projects, and it's no real surprise because it's really good!

Setting up Travis CI

To use Travis, you will need an account on GitHub. If you haven't got one, navigate to <https://github.com/> and register there. When you have a GitHub account, navigate to <https://travis-ci.org/> and click on **Sign in with GitHub**.

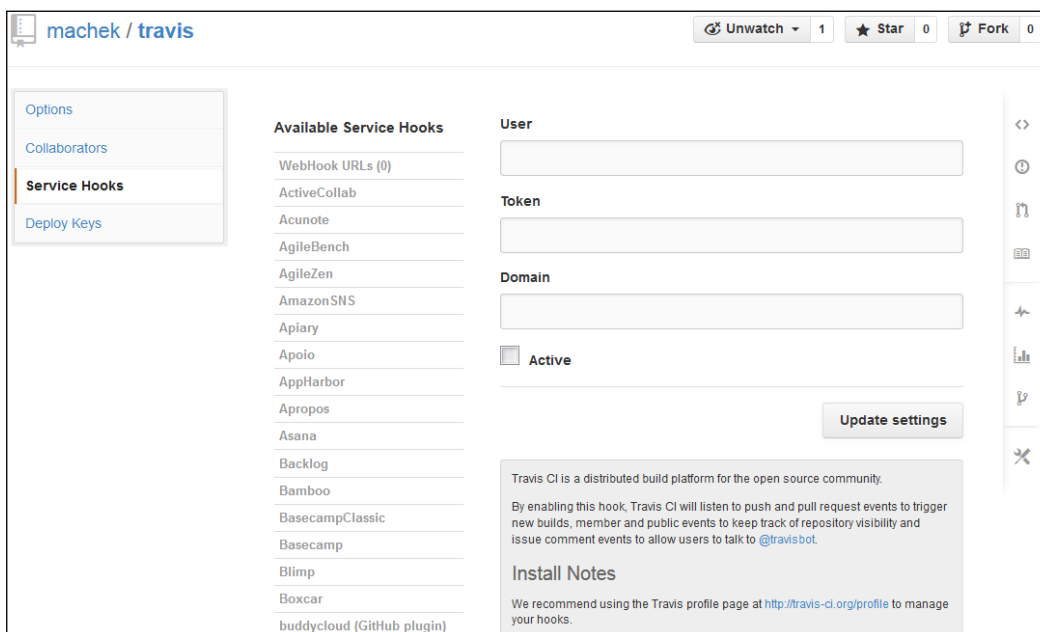


Continuous Integration

As you can see in the preceding screenshot, there will be a Travis application added to your GitHub account. This application will work as a trigger that will start a build after any change is pushed onto the GitHub repository. To configure the Travis project, you have to follow these steps:

1. You will be asked to allow Travis to access your account.
2. When you do this you will go back to the Travis site, where you will see a list of your GitHub repositories.
3. By clicking on **On/Off**, you can decide which project should be used by Travis.
4. When you click on a project configuration, you will be taken to GitHub to enable the service hook. This is because you have to run a build after every commit, and Travis is going to be notified about this change.
5. In the menu, search for Travis and fill in the details that you can find in your Travis account settings. Only the username and token are required, and the domain is optional.

For a demonstration, you can refer to my sample project, where there is just one test suite, and its purpose is to test how Travis works (navigate to <https://github.com/machek/travis>):



Using Travis CI

When you link your GitHub account to Travis and set up a project to notify Travis, you need to configure the project.

You need to follow the project setup in the same way that we did in the previous chapters. To have classes, you are required to have the test suites that you want to run, a bootstrap file, and a `phpunit.xml` configuration file.

You should try this configuration locally to ensure that you can run PHPUnit, execute tests, and make sure that all tests pass.

If you cloned the sample project, you will see that there is one important file: `.travis.yml`.

This Travis configuration file is telling Travis what the server configuration should look like, and also what will happen after each commit.

Let's have a look at what this file looks like:

```
# see http://about.travis-ci.org/docs/user/languages/php/ for more
# hints
language: php

# list any PHP version you want to test against
php:
  - 5.3
  - 5.4

# optionally specify a list of environments
env:
  - DB=mysql

# execute any number of scripts before the test run, custom env's
# are available as variables
before_script:
  - if [[ "$DB" == "mysql" ]]; then mysql -e "create database IF
    NOT EXISTS my_db;" -uroot; fi

# omitting "script:" will default to phpunit
script: phpunit --configuration phpunit.xml --coverage-text

# configure notifications (email, IRC, campfire etc)
notifications:
  email: "your@email"
```

Continuous Integration

As you can see, the configuration is really simple, and it shows that we need PHP 5.3 and 5.4, and a MySQL database to create a database, execute the PHPUnit with our configuration, and send a report to my e-mail address.

After each commit, PHPUnit executes all the tests. The following screenshot shows us an interesting insight into how Travis executes our tests and which environment it uses:

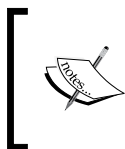
The screenshot displays a Travis CI build job #4.1, which has passed. The job details include the commit hash 7c721fce (master), the author Zdenek Machek, and the duration of 19 seconds. The build message is "cleaning up". Below the job details, a terminal window shows the execution steps:

```

1 Using worker: worker-linux-8-2.bb.travis-ci.org:travis-linux-20
2
3 $ export DB=mysql
4 $ git clone --depth=50 --branch=master git://github.com/machek/travis.git machek/travis
12 $ cd machek/travis
13 $ git checkout -qf 7c721fce26f4c449412668bad74620ab4b25f2d9
14 $ phpenv global 5.3
15 $ php --version
16 PHP 5.3.27 (cli) (built: Nov 18 2013 20:30:26)
17 Copyright (c) 1997-2013 The PHP Group
18 Zend Engine v2.3.0, Copyright (c) 1998-2013 Zend Technologies
19 with Xdebug v2.2.3, Copyright (c) 2002-2013, by Derick Rethans
20 $ composer --version
21 Composer version 348031cc7377de589f60f141b95c6fd0d42f3993 2013-11-15 13:04:24
22 $ if [[ "$DB" == "mysql" ]]; then mysql -e "create database IF NOT EXISTS my db;" -uroot; fi
23 $ phpunit --configuration phpunit.xml --coverage-text
24 PHPUnit 3.7.28 by Sebastian Bergmann.
25
26 Configuration read from /home/travis/build/machek/travis/phpunit.xml
27
28 ..
29
30 Time: 178 ms, Memory: 8.00Mb

```

You can view the build and the history for all builds.



Even though there are no real builds in PHP because PHP is an interpreted language and not compiled, the action performed when you clone a repository, execute PHPUnit tests, and process results is usually called a **build**.

Travis configuration can be much more complex, and you can run Composer to update dependency and much more. Just check the Travis documentation for PHP at <http://about.travis-ci.org/docs/user/languages/php/>.

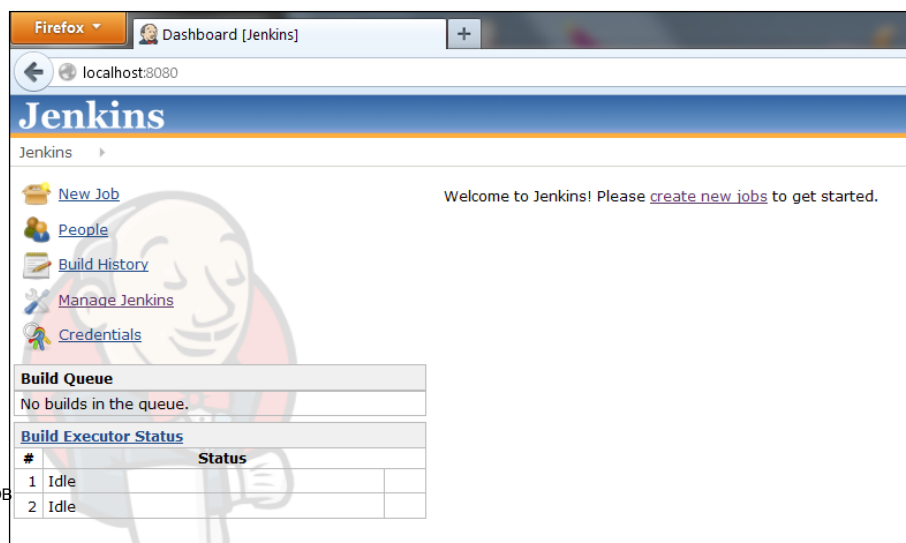
Using the Jenkins CI server

Jenkins is a CI server. The difference between Travis and Jenkins is that when you use Travis as a service, you don't have to worry about the configuration, whereas Jenkins is a piece of software that you install on your hardware. This is both an advantage and a disadvantage. The disadvantage is that you have to manually install it, configure it, and also keep it up to date. The advantage is that you can configure it in a way that suits you, and all of the data and code is completely under your control. This can be very important when you have customer code and data (for testing, never use live customer data) or sensitive information that can't be passed on to a third party.

The Jenkins project started as a fork of the Hudson project and is written in Java but has many plugins that suit a variety of programming languages, including PHP. In recent years, it has become very popular, and nowadays it is probably the most popular CI server. The reasons for its popularity are that it is really good, can be configured easily, and there are many plugins available that probably cover everything you might need.

Installation

Installation is a really straightforward process. The easiest method is to use a Jenkins installation package from <http://jenkins-ci.org/>. There are packages available for Windows, OS X, and Linux, and the installation process is well-documented there. Jenkins is written in Java, which means that Java or OpenJDK is required. After this comes the installation, as you just launch the installation and point it to where it should be installed, and Jenkins is listening on port 8080.



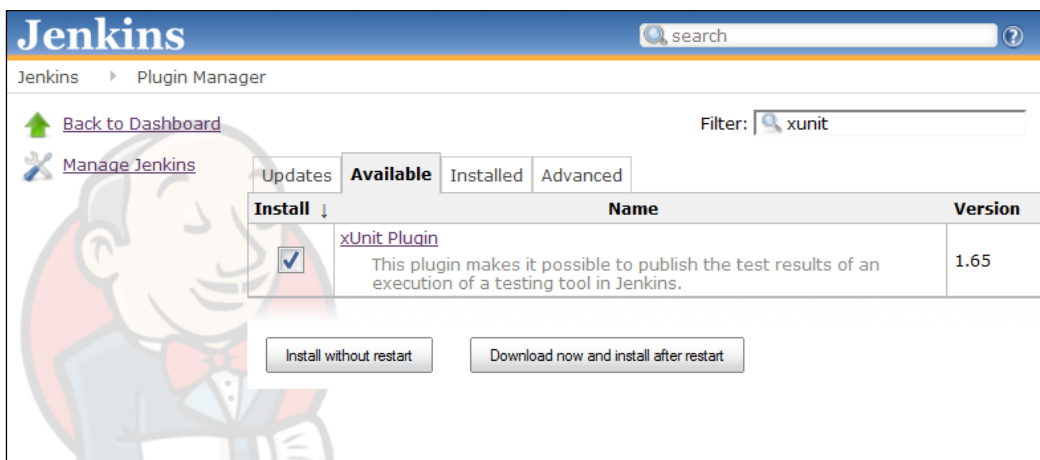
Continuous Integration

Before we move on to configure the first project (or job in Jenkins terminology), we need to install a few extra plugins. This is Jenkins' biggest advantage. There are many plugins and they are very easy to install. It doesn't matter that Jenkins is a Java app as it also serves PHP very well.

For our task to execute tests, process results, and send notifications, we need the following plugins:

- **Email-ext:** This plugin is used to send notifications
- **Git or Subversion:** This plugin is used to check the code
- **xUnit:** This plugin is used for processing the PHPUnit test results
- **Clover PHP:** This plugin is used for processing the code coverage

To install these plugins, navigate to **Jenkins | Manage Jenkins | Manage Plugins** and select the **Available** tab. You can find and check the required plugins, or alternatively use the search filter to find the one you need:



For e-mails, you might need to configure the STMP server connection at **Manage Jenkins | Configure System | E-mail notification** section.

Usage

By now, we should have installed everything that we need, and we can start to configure our first simple project. We can use the same simple project that we used for Travis. This is just one test case, but it is important to learn how to set up a project. It doesn't matter if you have one or thousands of tests though, as the setup is going to be the same.

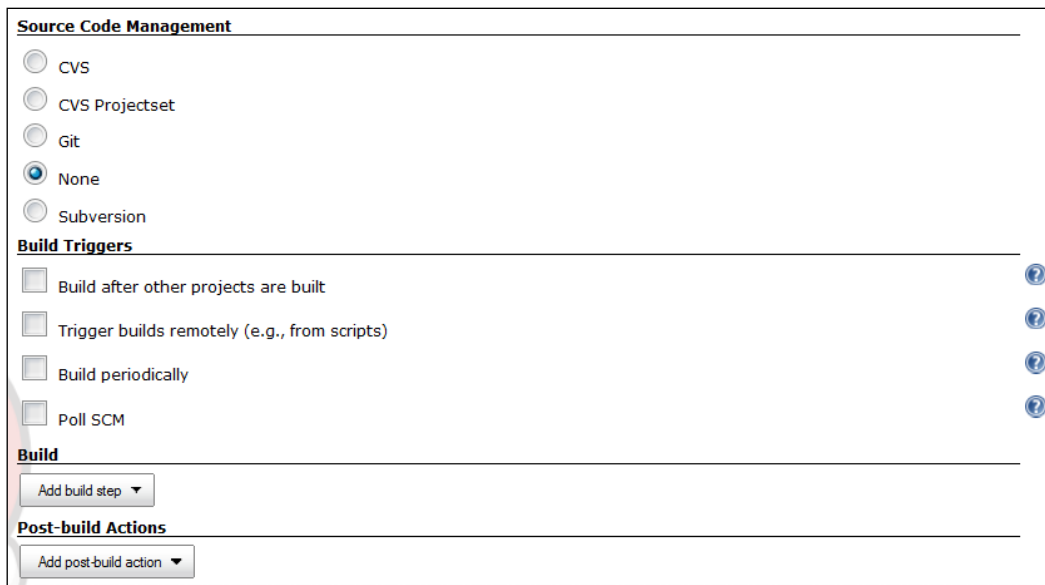
Creating a job

The first step is to create a new job. Select **New Job** from the Jenkins main navigation window, give it a name, and select **Build a free-style software project**. After clicking on **OK**, you get to the project configuration page.

The most interesting things there are listed as follows:

- **Source Code Management:** This is where you check the code
- **Build Triggers:** This specifies when to run the build
- **Build:** This tests the execution for us
- **Post-build Actions:** This publishes results and sends notifications

The following screenshot shows the project configuration window in Jenkins CI:



The screenshot displays the Jenkins project configuration interface, organized into four main sections:

- Source Code Management:** This section contains five radio button options: CVS, CVS Projectset, Git, None (which is selected), and Subversion.
- Build Triggers:** This section contains four checkbox options: 'Build after other projects are built', 'Trigger builds remotely (e.g., from scripts)', 'Build periodically', and 'Poll SCM'. Each checkbox has a corresponding help icon (a question mark in a circle) to its right.
- Build:** This section features a single button labeled 'Add build step' with a small downward arrow.
- Post-build Actions:** This section features a single button labeled 'Add post-build action' with a small downward arrow.

Source Code Management

Source code management simply refers to your version control system, path to the repository, and the branch/branches to be used. Every build is a clean operation, which means that Jenkins starts with a new directory where the code is checked.

Build Triggers

Build triggers is an interesting feature. You don't have to use it and you can start to build manually, but it is better to specify when a build should run. It can run periodically at a given interval (every two hours), or you can trigger a build remotely.

One way to trigger a build is to use post commit hooks in the Git/SVN repository. A post commit hook is a script that is executed after every commit. Hooks are stored in the repository in the `/hooks` directory (`.git/hooks` for Git and `/hooks` for SVN). What you need to do is create a post-commit (SVN) or post-receive (Git) script that will call the URL given by Jenkins when you click on a **Trigger remotely** checkbox with a secret token:

```
#!/bin/sh
wget
  http://localhost:8080/job/Sample_Project/build?
  token=secret12345ABC -O /dev/null
```

After every commit/push to the repository, Jenkins will receive a request to run the build and execute the tests to check whether all of the tests work and that any code change there is not causing unexpected problems.

Build

A build is something that might sound weird in the PHP world, as PHP is interpreted and not compiled; so, why do we call it a build? It's just a word. For us, it refers to a main part of the process — to execute unit tests.

You have to navigate to **Add a build step** — click on either **Execute Windows batch command** or **Execute shell**. This depends on your operating system, but the command remains the same:

```
phpunit --log-junit=result.xml --coverage-clover=clover.xml
```

This is simple and outputs what we want. It executes tests, stores the results in the JUnit format in the file `result.xml`, and generates code coverage in the clover format in the file `clover.xml`.

I should probably mention that PHPUnit is not installed with Jenkins, and your build machine on which Jenkins is running must have PHPUnit installed and configured, including PHP CLI.

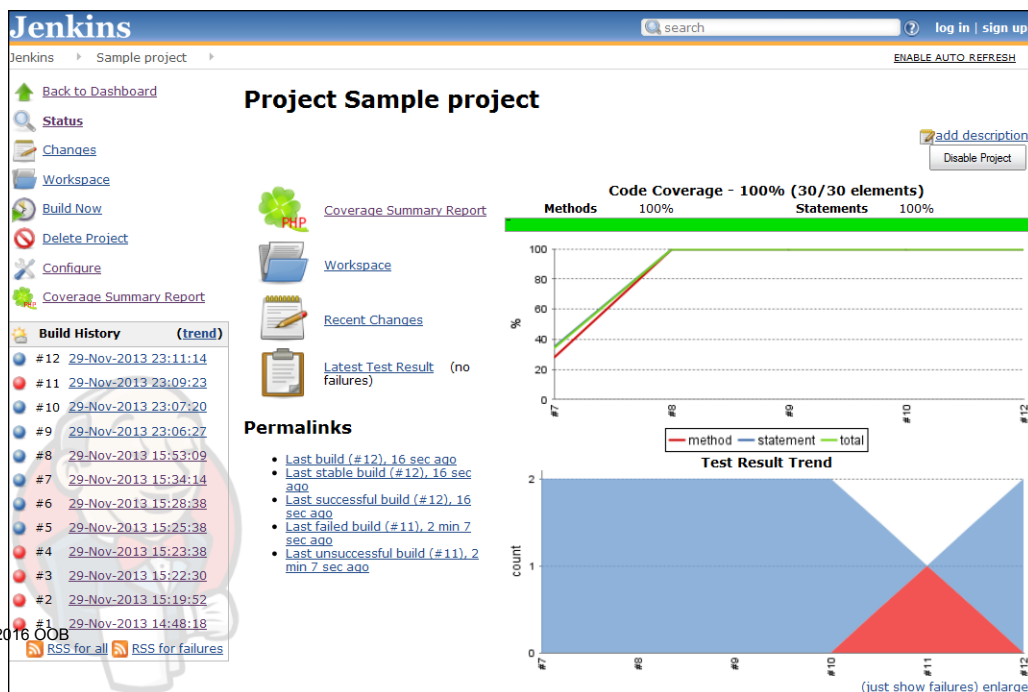
Post-build Actions

In our case, there are three post-build actions required. They are listed as follows:

- **Process the test result:** This denotes whether the build succeeded or failed. You need to navigate to **Add a post-build action | Publish Junit test result report** and type `result.xml`. This matches the switch `--log-junit=result.xml`. Jenkins will use this file to check the tests results and publish them.
- **Generate code coverage:** This is similar to the first step. You have to add the **Publish Clover PHP Coverage report** field and type `clover.xml`. It uses a second switch, `--coverage-clover=clover.xml`, to generate code coverage, and Jenkins uses this file to create a code coverage report.
- **E-mail notification:** It is a good idea to send an e-mail when a build fails in order to inform everybody that there is a problem, and maybe even let them know who caused this problem and what the last commit was. This step can be added simply by choosing **E-mail notification action**.

Results

The result could be just an e-mail notification, which is handy, but Jenkins also has a very nice dashboard that displays the current status for each job, and you can also see and view the build history to see when and why a build failed.



Continuous Integration

A nice feature is that you can drill down through the test results or code coverage and find more details about test cases and code coverage per class.

If you're not satisfied with the results, you can use the PHPUnit HTML coverage report that we created in *Chapter 5, Running Tests from the Command Line*, and navigate to **Post build action** | **Publish HTML reports** to use the report created with PHPUnit.



To make testing even more interesting, you can use Jenkins' The Continuous Integration Game plugin. Every developer receives positive points for written tests and a successful build, and negative points for every build that they broke. The game leaderboard shows who is winning the build game and writing better code.

Using the Xinc PHP CI server

Xinc is a CI server written in PHP. What is interesting about this server is that it's in PHP, which means it's going to be a lightweight solution compared to Java applications, and when you have to pay for cloud service, then the amount of memory matters. With Xinc, 512 MB of memory might be enough, whereas for Java you will need at least 1 GB of memory. Another advantage is that it's PHP, so if there is any problem or you want to tweak or extend it, then it's easy.

A disadvantage is that this system is not used as widely as the previous two solutions, and its configuration might also look a bit confusing. Yet, it has pretty good documentation, and so it shouldn't be too difficult to overcome these problems.

Xinc has two parts. The first part is Xinc running as a background process and checking repositories for changes and starting builds, while the second part is a web application that provides an interface to the test results.

Installation

The recommended installation method is to use a PEAR installer. There are a few required dependencies, so manual installation might be a bit tricky.

For installation, you need to discover a few PEAR channels and then run the installation:

```
>pear config-set auto_discover 1
>pear channel-discover pear.phpunit.de
>pear channel-discover pear.elektrischeslicht.de
```



```
>pear channel-discover components.ez.no
>pear install --alldeps xinc/Xinc
```

After installation, you need to run the following post installation script:

```
>pear run-scripts xinc/Xinc
```

This script gives you a few options about where and how Xinc will be installed. You can press the *Enter* key to use the default installation:

```
root@precise64: /home/vagrant# pear run-scripts xinc/Xinc
Including external post-installation script "/usr/share/php/Xinc/Postinstall/Nix
.php" - any errors are in this script
Inclusion succeeded
running post-install script "Xinc_Postinstall_Nix_postinstall->init()"
init succeeded
1. Directory to keep the Xinc config files                : /etc/xinc
2. Directory to keep the Xinc Projects and Status information : /var/xinc
3. Directory to keep the Xinc log files                   : /var/log
4. Directory to install the Xinc start/stop daemon       : /etc/init.d
5. Directory for xinc's temporary files                   : /usr/share/php/
data/Xinc/tmp
6. Do you want to install the SimpleProject example      : yes
7. Directory to install the Xinc web-application         : /var/www/xinc
8. IP of Xinc web-application                           : 127.0.0.1
9. Port of Xinc web-application                         : 8080

1-9, 'all', 'abort', or Enter to continue: █
```

After installation, you will get instructions to enable Xinc to operate on your machine, and they will look similar to this:

Xinc installation complete.

- Please include /etc/xinc/www.conf in your apache virtual hosts.
- Please enable mod-rewrite.
- To add projects to Xinc, copy the project xml to /etc/xinc/conf.d
- To start xinc execute: `sudo /etc/init.d/xinc start`

UNINSTALL instructions:

- `pear uninstall xinc/Xinc`
- run: `/usr/bin/xinc-uninstall` to cleanup installed files

[OK] Saved ini settings

Install scripts complete

Continuous Integration

After installation, it is a good idea to check whether Xinc actually works because if there is a problem, it might be really tricky to debug what's going on. You can check it by running the following line of code:

```
>/usr/bin/xinc
```

One of the common problems that you can see now is this error message:

```
PHP Warning:  require(Xinc.php): failed to open stream: No such file
or directory in /usr/bin/xinc on line 32
```

```
PHP Stack trace:
```

```
PHP    1. {main}() /usr/bin/xinc:0
```

```
PHP Fatal error:  require(): Failed opening required 'Xinc.php'
(include_path='.:root/pear/share/pear') in /usr/bin/xinc on line
32
```

```
PHP Stack trace:
```

```
PHP    1. {main}() /usr/bin/xinc:0
```

This problem has a simple solution; check whether in your `php.ini` file, PEAR is added to `include_path`.

After these steps, you will have Xinc running as a background task and listening on port 8080 as a web application (you can change it in `www.conf` if you don't like it), and you will have also installed a sample project.

Usage

After installation, we will focus on the aforementioned Simple Project. The configuration file is located at `/etc/xinc/conf.d/simpleproject.xml`. Xinc uses Phing (for more information, refer to <http://www.phing.info>). This is a PHP clone of Ant—a building tool that drives processes. You can think of it as a script that says run this and then this. It might sound a bit strange to learn that it uses XML, but when you get used to it, you will realize it's quite an interesting way to go about things.

Ant is far more popular than Phing, but the logic and structure is the same or very similar, and Phing gives you an advantage as there is an option to extend it with PHP if required.

Let's take this simple project and modify it for the example that we used earlier by following the same steps:

1. Check the Git repository.

3. Publish the results.
4. Send notifications.

Almost everything is in place for a sample project, and there are two important files at the location `/etc/xinc/conf.d/sampleproject.xml`:

```
<?xml version="1.0"?>
<xinc>
  <project name="SimpleProject">
    <configuration>
      <setting name="loglevel" value="1"/>
      <setting name="timezone" value="US/Eastern"/>
    </configuration>
    <property name="dir"
value="/var/xinc/projects/SimpleProject"/>
    <cron timer="*/4 * * * *"/>
    <modificationset>
      <buildalways/>
    </modificationset>
    <builders>
      <phingBuilder buildfile="${dir}/build.xml"
target="build"/>
    </builders>
    <publishers>
      <phpUnitTestResults
file="${dir}/report/logfile.xml"/>
      <onfailure>
        <email to="root"
subject="${project.name} build
${build.number} failed"
message="The build failed."/>
      </onfailure>
      <onsuccess>
        <phingPublisher
buildfile="${dir}/publish.xml" target="build"/>
        <artifactspublisher
file="${dir}/publish.xml"/>
        <artifactspublisher
file="${dir}/publish.xml"/>
        <deliverable file="${dir}/builds/release-
${build.label}.tar.gz" alias="release.tar.gz"/>
      </onsuccess>
      <onrecovery>
        <email to="root"
subject="${project.name} build
${build.number} was recovered"
message="The build passed after having
failed before."/>
      </onrecovery>
    </publishers>
  </project>
</xinc>
```

Continuous Integration

```

        </onrecovery>
    </publishers>
</project>
</xinc>

```

This is a project configuration file that checks for modifications every 4 minutes. Run the build, publish the results, and send notifications when the build fails or recovers. The main configuration is in the following line:

```
<phingBuilder buildfile="${dir}/build.xml" target="build"/>
```

The preceding line says the build configuration file is sitting inside the project directory and is called `build.xml`. Let's have a look at what's there:

```

<?xml version="1.0"?>
<project name="Simple Project Build File"
  basedir="/var/xinc/projects/SimpleProject" default="build">
  <property name="report.dir"
    value="${project.basedir}/report"/>
  <target name="build" depends="prepare, test, tar, generate-
    report">

    </target>
    <target name="prepare">
      <mkdir dir="${report.dir}"/>
    </target>
    <target name="tar">
      <tar destfile="${project.basedir}/release-
        ${xinc.buildlabel}.tar.gz" compression="gzip">
        <fileset dir=".">
          <include name="index.php" />
          <include name="Page.php" />
        </fileset>
      </tar>
    </target>
    <target name="test">
      <phpunit haltonfailure="true" printsummary="true">
        <batchtest>
          <fileset dir=".">
            <include name="*Test.php"/>
          </fileset>
        </batchtest>
        <formatter type="xml" todir="${report.dir}"
          outfile="logfile.xml"/>
      </phpunit>
    </target>
    <target name="generate-report">

```

```

        <phpunitreport infile="${report.dir}/logfile.xml"
            styledir="resources/xsl" todir="report"
            format="noframes"/>
    </target>
</project>

```

This is actually a Phing script. In Phing (Ant) terminology, tasks are called **targets**, and the dependency specifies what tasks need to be executed. For this build, the tasks that are going to be executed are prepare, test, tar, and generate-report targets.

This is what we need with one exception; we need to check the Git repository. The first step is manual, but it could be part of the build. We create/code the directory and clone the GitHub repository we used for previous tests:

```
git clone https://github.com/machek/travis
```

To be able to use Git with Phing, we have to install the Git plugin:

```
>pear install VersionControl_Git-alpha
```

Now, we need to slightly modify the build script to pull from the Git repository and execute our tests:

```

<?xml version="1.0"?>
<project name="Simple Project Build File"
    basedir="/var/xinc/projects/SimpleProject" default="build">
    <property name="report.dir"
        value="${project.basedir}/report"/>
    <property name="code.dir" value="${project.basedir}/code"/>
    <target name="build" depends="prepare, gitpull, test, tar,
        generate-report">

        </target>
    <target name="prepare">
        <mkdir dir="${report.dir}"/>
        <mkdir dir="${code.dir}"/>
    </target>
    <target name="gitpull">
        <echo msg="Getting latest code from ${git.repo}" />
        <gitpull gitPath="git" repository="${code.dir}" all="true" />
    </target>
    <target name="tar">
        <tar destfile="${project.basedir}/release-
            ${xinc.buildlabel}.tar.gz" compression="gzip">
            <fileset dir=".">
                <include name="index.php" />

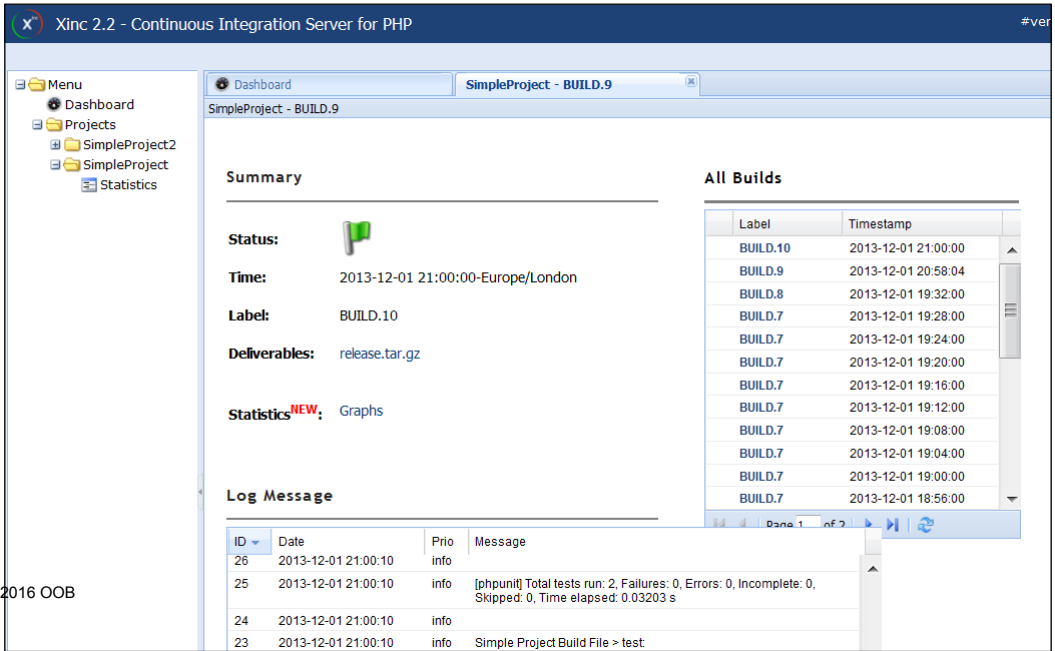
```

Continuous Integration

```

        <include name="Page.php" />
    </fileset>
</tar>
</target>
<target name="test">
    <phpunit haltonfailure="true" printsummary="true"
        configuration="${code.dir}/phpunit.xml">
        <batchtest>
            <fileset dir="${code.dir}/tests">
                <include name="*Test.php"/>
            </fileset>
        </batchtest>
        <formatter type="xml" todir="${report.dir}"
            outfile="logfile.xml"/>
    </phpunit>
</target>
<target name="generate-report">
    <phpunitreport infile="${report.dir}/logfile.xml"
        styledir="resources/xsl" todir="report"
        format="noframes"/>
</target>
</project>
```

There is one extra target, `gitpull`, now, and the target tests were also changed to use our tests. This is all that you need to set up a project. When you access the Xinc dashboard served by Apache, you should see a result similar to the following:



Summary

A continuous integration server is something you really need. You can run PHPUnit tests from a command line, but it's much nicer when it happens automatically, as you have a history of test results and code coverage, and you can notify developers when somebody breaks the tests. It's like a cherry on top, but there is something satisfying when you see that all of your hard work makes sense and you can see its results. Drill through the test results and see the code coverage for each class and method.

We saw three examples, which are probably the most interesting solutions in the PHP world. Travis is very easy to configure and use as a service for open source projects. The king of CI servers is Jenkins, with tons of plugins and easy configuration, and the Xinc CI server is written in PHP. It might depend on your project, but you should try to choose the right one for you. Of course, you can do much more than just run unit tests. You might need a Selenium Server, and then the question is whether your CI server will also host a Selenium Server. You can try to find out more about your code with tools such as PHP CodeSniffer in order to detect if your developers are following coding standards. There is so much more that you can do with CI servers, so just give it a try. You will love them!

In the next and last chapter, we are going to check some PHPUnit alternatives. Even though PHPUnit is a very good tool, there are a few interesting projects available that take a slightly different approach with respect to testing. To gain a complete picture about PHP and writing tests, we are going to look at a few of them, and you will see what **Behavior-Driven Development (BDD)** means.

