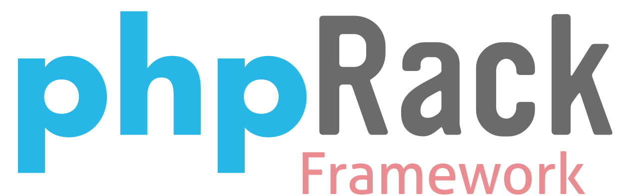
Integration Testing with



by Yegor Bugayenko

Unit tests plus continuous integration give you a guarantee that most of your source code defects will be caught before they reach end users. Unit tests verify the code beforehand and a continuous integration server informs you if a defect is found, prior to deployment. It's a proven mechanism used by many teams striving for quality, but who can guarantee that if a defect is found after deployment, it is reported immediately before disappointed end users find it? Integration tests can achieve this.

REQUIREMENTS

PHP: 5.2+

Other Software:
- phpRack 0.1
- Phing 2.4.2

Related URLs:
- phpRack homepage: http://www.phprack.com
- Phing: http://www.phing.info
- PHPUnit: http://www.phpunit.de

Licensed to 46625 - Yegor Bugayenko (egor@technoparkcorp.com)

What is Integration Testing?

In short, “Integration Testing” is similar to unit testing, however, the testing subject is not the software itself, but its deployment environment. With integration tests, you’re verifying whether your product is correctly installed and is ready for end users. This article explains the concept and its implementation in phpRack, the open-source framework. phpRack is written in PHP, but can be used for the integration testing of products written in any other web-oriented language, including Java, Python, and Ruby.

Let’s start with a simple example. Imagine you have developed a web application and prepared it for delivery to your customer. You created complete and detailed installation instructions where you explained the requirements for the server-side system software configuration. Despite your efforts, very soon you receive a complaint from the customer: “The system is full of bugs, and we can’t make it work!” Yet you know for sure that the product is properly tested and works on your local machine. You understand that the problem is in the installation, within the customer’s server environment. How do you explain to the customer where the problem is and how to fix it?

One option is to do the work yourself. Ask the customer for root access to the server, debug the software there, find the problems, fix them, and make the customer happy. This solution is very common, but is also a very costly and risky method. Firstly, you don’t know how much time this procedure might take. Secondly, you might unintentionally break other products on the same server. Thirdly, you will become responsible for the server and this installation forever. The list of potential problems is endless. The bottom line is that this manual approach, although very common, is the worst option.

Integration tests are a great alternative. You create small supplementary software modules that are executed on the production server and validate critical environmental details. Such details might include the version of PHP, availability of PHP extensions, version of MySQL, database access permissions, accessibility of network ports, read/write access permissions of log files and directories. Some of my products have 50 or more different integration tests. In addition to the integration tests, you need software that will start all of them together and display a summary report in a user-friendly format. Then, you can show this report to the customer, indicating the problems with the server. Once they are fixed, your product works as if it were installed on your local machine. Easy!

phpRack simplifies this work, and allows you to write integration tests quickly and easily. phpRack is the first product of its kind (at least, I didn’t find any similar testing frameworks on the market). phpRack is a free open-source product. I’ll explain further how to start using it in only five minutes, and how to write different integration tests using it.

LISTING 1

```
1. $phpRackConfig = array(
2.     'dir' => './rack-tests',
3.     'auth' => array(
4.         'username' => 'myusername',
5.         'password' => 'mypassword',
6.         'onlySSL'  => false,
7.     ),
8.     'htpasswd' => './.htpasswd',
9.     'notify' => array(
10.        'email' => array(
11.            'transport' => array(
12.                'class'    => 'smtp',
13.                'host'     => 'smtp.gmail.com',
14.                'port'     => '25',
15.                'tls'      => false,
16.                'username' => 'myusername',
17.                'password' => 'mypassword',
18.            ),
19.            'recipients' => 'admin@example.com'
20.        ),
21.    ),
22. );
23. include './phpRack/bootstrap.php';
```

How to Install phpRack and Create Your First Test

All you need to install phpRack on your server or shared hosting account is to download it from the phpRack site (see Related URLs), and copy it to the server. Then create a PHP script, and write at least one integration test. The script must be available to your HTTP server and accessible from the web. For example:

```
// file name is phprack.php
$phpRackConfig = array('dir' => './rack-tests');
include './phpRack/bootstrap.php';
```

Firstly, you define a single global variable **$phpRackConfig**, telling the framework where your integration tests are located (a more detailed example of this can be seen in Listing 1, and we’ll return to this variable later). Next, you **include** the framework bootstrapping mechanism. We assume that you uploaded the framework to a directory called *phpRack*.

Licensed to 46625 - Yegor Bugayenko (egor@technoparkcorp.com)

Next, you create a simple integration test, placing it into *rack-tests/MyTest.php*. For example:

```
class MyTest extends phpRack_Test
{
    public function testPhpVersionIsCorrect()
    {
        $this->assert->php->version->atLeast('5.2');
    }
}
```

That's it, configuration is done. Try to open your *phprack.php* file in a browser, and see the result. You should see something similar to the output seen at http://www.phprack.com/phprack.php.

The integration test we just created does not do much. It validates that the server has PHP version 5.2+ installed, not earlier. This looks simple, but an incompatible PHP version is one of the most common sources of error during deployment. Your local machine has PHP 5.2, and you assume that the customer's server also runs the same version. In many cases, such an assumption is false. Moreover, the server can be modified at any time. Once the integration test is written, neither you nor your customer should forget that the product requires PHP 5.2.

By the way, if you're familiar with unit testing, this approach will look similar. Later, we will discuss how to use phpRack in your continuous integration environment, together with Phing.

Where, When and Why You Might Need Integration Tests

Let's review a number of use cases where integration tests would have saved you time and effort and spared the nerves of your customers and end-users. Some of them might look familiar, and others are less obvious, but I encourage you to use as many such tests in your application as possible. Imagine that an integration test is your representative at the remote location, asking questions and validating answers received:

PHP Configuration The installed version of PHP must be 5.2 or higher. The **simplexml**, **fileinfo**, and **soap** extensions are required. The **short_open_tag** configuration setting must be set to **ON**.

Disc and System Files PHP scripts in the web environment must have write access to log files. The disc storage must have enough space for at least 100Mb of free space, for example.

Databases and DB Servers The MySQL server version must be 5.0 or higher. The server must allow access to a pre-configured user. The database must exist on the server and must have a table named **project**.

Source Code All source files are valid according to **phplint**. This also applies to XML and JS files.

Cloud Resources Cloud storage is accessible in the network. Third-party network resources are available, ports are open and login credentials work.

PEAR Packages Required PEAR packages are installed, and their versions are equal to or higher than required.

System Performance Performance of the production platform is higher than the minimum required for the product. The file system engine is fast enough.

Third-Party Software Required software components are installed and configured as required (for example, an SMTP server).

This is just a sample list of situations when integration tests could be very helpful. All validations could be done manually, but imagine how much time it would take to do all of them every time you have a strange configuration problem on the production server. Furthermore, it's absolutely impossible to do them manually, say, every hour.

Format of Assertions is Simple

Every phpRack integration test is a small PHP class with a number of methods, whose names start with the **test** prefix. All phpRack instruments are accessible inside these methods by means of the assert shortcut. For example:

```
public function testSomeAssertions()
{
    $this->assert
        ->isTrue(is_readable('log.txt'))
        ->isTrue(ini_get('short_open_tag'));
}
```

By design, every assertion returns **$this** which allows us to use a fluent interface for a number of consecutive calls, as in the example above.

It is not mandatory to use assertions in all integration tests. For instance, if you are retrieving the content of a log file and showing it to the end user, you don't assert anything. Such a test doesn't have any result and will be ignored in batch mode.

phpRack has a library of assertions which make the design of integration tests easy and convenient. For example:

Licensed to 46625 - Yegor Bugayenko (egor@technoparkcorp.com)

```
public function testSomeAssertions()
{
    $this->assert->disc->file->isReadable('log.txt');
    $this->assert->php->ini('short_open_tag');
}
```

In general, you are encouraged to use the assertions library instead of writing your own mechanisms. A full reference to the library is available online at http://www.phprack.com/.

Similar to unit testing frameworks, in phpRack you have the ability to override two "magic" methods in any integration test: **setUp()** and **tearDown()**. These are respectively called before and after every test method. These magic methods are very useful when you want to instantiate and configure some supplementary variables. You can't override a constructor, but you can use the virtual method **_init()** if and when you want to pre-configure your integration test. As seen in Listing 2, such a method is a good place for setting the AJAX-related options of a test.

Continuous Integration with phpRack

First and foremost, an integration test is a server-side script that automates your routine manual testing operations in the production environment. A bigger benefit is that when you are using integration tests as part of your continuous integration cycle, phpRack can run integration tests not only one-by-one, but in a batch, reporting to you the overall result. If all integration tests return successful, your build scenario can continue, otherwise it fails. Let's review an example (see Listing 3).

A typical build scenario in a PHP project could include the following steps (exactly in the given order):

- validation of source code syntax using PHPLint
- unit testing with PHPUnit
- deployment by FTP

One of the best tools to automate this process is Phing, which allows you to specify the scenario just once in an XML config file and execute it automatically when necessary (see Listing 3 for an example).

LISTING 2

```
1.  class LoadTest extends PhpRack_Test
2.  {
3.      protected function _init()
4.      {
5.          $this->setAjaxOptions(
6.              array(
7.                  'reload' => 5,
8.              )
9.          );
10.     }
11.     public function testServerStatus()
12.     {
13.         $this->assert->shell
14.             ->exec('whoami 2>&1', '/apache/')
15.             ->exec('df 2>&1')
16.             ->exec('free -t -m 2>&1')
17.             ->exec('uptime 2>&1')
18.             ->exec(
19.                 'ps o "%cpu %mem time command" ax'
20.                 . ' | sort -k 1 -r'
21.                 . ' | head -5'
22.             );
23.     }
24.     public function testViewLog()
25.     {
26.         $this->assert->disc->file->tail(
27.             '/home/me/log.txt',
28.             10
29.         );
30.     }
31. }
```

Phing ensures that every consecutive step in the build scenario is started if, and only if, a previous one finished with success. Thus, if PHPLint validation found a syntax error in your code, Phing would not start unit testing and the product would not be deployed.

The rule of thumb is that each and every test you add to the build scenario **before deployment** adds quality to the product. In other words, you should

LISTING 3

```
1.  <?xml version="1.0" ?>
2.  <project name="MyProject" basedir="." default="main">
3.      <target name="main" depends="lint, test, deploy">
4.      </target>
5.      <target name="lint">
6.          <phplint haltonfailure="yes" level="verbose">
7.              <fileset dir="${project.basedir}">
8.                  <include name="**/*.php"/>
9.                  <include name="**/*.phtml"/>
10.                 <exclude name=".svn/**"/>
11.             </fileset>
12.         </phplint>
13.     </target>
14.     <target name="test">
15.         <includepath classpath="${project.basedir}"/>
16.         <includepath classpath="${project.basedir}/test/"/>
17.             <phpunit haltonerror="yes" haltonfailure="yes">
18.                 <batchtest>
19.                     <fileset dir="${project.basedir}/test">
20.                         <include name="**/*Test.php"/>
21.                         <exclude name="**/Abstract*.php"/>
22.                         <exclude name=".svn/**"/>
23.                     </fileset>
24.                 </batchtest>
25.             </phpunit>
26.     </target>
27.     <target name="deploy">
28.         <ftpdeploy username="deployer" password="K9Pw3F"
29.             host="ftp.example.com">
30.                 <fileset dir="${project.basedir}/src">
31.                     <include name="**/*"/>
32.                     <exclude name=".svn/**" />
33.                 </fileset>
34.         </ftpdeploy>
35.     </target>
36. </project>
```

Licensed to 46625 - Yegor Bugayenko (egor@technoparkcorp.com)

validate the product before it goes to the production environment. Once it's there, your users will validate it, and blame you for errors.

The next step in automation is to use continuous integration (CI) software such as CruiseControl or Hudson, which can be configured to run Phing as soon as you make any change to the source code. Once you make a new commit to your source code repository, CI software updates its own copy of the repository and starts Phing. If your change didn't break any quality validators (syntax, unit tests, etc.) that go before deployment, the product will be successfully deployed. Otherwise, you will receive a notification, and CI will try to run Phing again some time later, perhaps after your next commit or on a schedule.

Integration tests are different from all other tests you might run before deployment. Integration tests must be executed **after deployment**, when the product is already in the production environment. If any problem is found, Phing must report a failure of the entire build scenario. Thus, your build scenario will looks like this:

- PHPLint
- unit testing with PHPUnit
- deployment by FTP
- integration testing with phpRack

Integration tests are run on the production server, and in order to execute them, Phing must make an HTTP request to phpRack on the server. phpRack will execute all integration tests, collect their results in a complete report and respond to Phing with it. Simple regular expression matching will tell Phing whether this report means success or failure. This is an example **<target>** for our *build.xml* file, which hooks it to phpRack:

```
<target name="phpRack">
   <http-request
      url="http://phprack.com/phprack.php?suite"
      authUser="myusername" authPassword="mypassword"
      responseRegex="/PHPRACK SUITE: OK/" />
</target>
```

It is assumed that the continuous integration cycle never stops, even when the product is not actively developed but is in its maintenance phase.

More Than Just Testing

phpRack is designed as a server-side testing engine and a web front end. The front end communicates with the testing engine by means of AJAX calls, starting/stopping tests and indicating their results. In addition, you can alter the behavior of the front end in order to get much more functionality than just testing.

Consider the example in Listing 2. This is an integration test with configuration instructions inside a virtual method **_init()**. Setting **reload** to five, we instruct phpRack to restart the test every five seconds and refresh its result on a web page. Thus, with just one option we've built a simple server monitoring console.

The code inside **testServerStatus()** will execute shell commands one-by-one and show their result. No testing is done here, just delivery of text from server to web front end. Only one call to `exec()` performs testing - **exec('whoami')**. This call has a second parameter, a regular expression, which you may have already noticed. phpRack will try to match the result of shell command execution with the regular expression provided, and if they don't match, the test will fail.

phpRack simplifies this work, and allows you to write integration tests quickly and easily.

The method **testViewLog()** doesn't test anything, but retrieves and delivers the last ten lines of the log file to the web front end. Since the entire integration test **LoadTest** will be reloaded by the web front end every five seconds, you will see the latest ten lines of the log on-the-fly. The same result can be achieve with a phpRack analog of the Unix command **tailf**:

```
class LogTest extends phpRack_Test
{
   public function testViewLog()
   {
      $this->assert->disc->file->tailf('log.txt');
   }
}
```

Licensed to 46625 - Yegor Bugayenko (egor@technoparkcorp.com)

The **tailf()** assertion will keep the latest twenty-five lines of the file visible on web, and will refresh them every second. This, as with all parameters, is configurable.

Complete Configuration Manual

As said before, to configure phpRack for your product, you should create a single *phprack.php* file where you include *bootstrap.php* from phpRack. There is one global variable: **$phpRackConfig** used in order to pass configuration parameters to the framework. Listing 1 has an example of your **phprack.php** file.

dir is the only mandatory parameter in this array. Its value should contain a relative or an absolute path to the directory with integration tests. A relative path should start with double dot (..) and will be resolved in relation to the location of the *phprack.php* file.

The **auth** parameter enables you to protect integration tests against public access. You can provide **username** and **password** or you can use the **htpasswd** parameter in order to provide a list of login credentials. The format of the file is the same as in Apache Server: username, colon, and the MD5 hash of the password. Instead of a file, you can provide an array of usernames and passwords without MD5 hashing. For example:

```
$phpRackConfig = array(
   'htpasswd' => array(
      'john' => 'jf7mF4',
      'alex' => 'Y6rT5p',
   )
);
```

The **auth/onlySSL** parameter explicitly instructs phpRack that only SSL encrypted web connections are allowed.

The **notify** parameter instructs phpRack to notify you when integration tests fail. For example, the **email** notifier will send a summary report by email if and when a problem is found. Notification is sent only when tests are executed in a batch mode, not individually.

For email notification, the **recipients** parameter may contain either one email address or an array of email addresses that will receive summary reports. The **transport** parameter may be omitted if your server allows you to send emails using the standard PHP mail() function (through **sendmail**).

Future Development of the phpRack Framework

The phpRack framework, in its first version, is already a powerful tool for a project of almost any size. However, there are a number of features we are going to release in the next versions, including:

Test Suites - To simplify the process of test design, we will introduce a library of test suites, which will provide ready-to-use collections of tests for certain purposes. For example, the ServerHealth suite will retrieve and display all available information about server load, check server performance, and detect server overload problems.

Adapters - We will develop adapters for different database servers, including non-relational ones and for different types of network/cloud resources, including Amazon, Rackspace and Azure. We will also create translators for different file types, which will enable on-the-fly retrieval of file properties.

Notifiers - Different notifiers will be developed, which will allow you to stay informed by SMS, via your Twitter account, on IRC or through any custom-configured XML RPC or SOAP server.

phpRack is developed by a small distributed team of PHP engineers. You can join our team, or if you have ideas or suggestions about future features, please email us at team@phprack.com.

Conclusion

Integration testing is a powerful tool for a web project of any size. Similar to unit testing, the more time you spend writing integration tests, the more time you save with debugging and installation. Integration testing is a mandatory mechanism if a product is likely to exist in multiple installations, or its production environment might be changed from time to time.

phpRack is an open-source software, which enables fast and easy development of integration tests and their use within continuous integration environment.

YEGOR BUGAYENKO is the lead architect of phpRack Framework and a proud holder of the ZCE, ZFCE and PMP certificates. He is also the director and co-founder of TechnoPark Corp., a custom software development company specializing in complex and distributed web applications.

Licensed to 46625 - Yegor Bugayenko (egor@technoparkcorp.com)